6:50s A Peek at Parallel Processing from an Applications Perspective

Massachusetts Institute of Technology
June 24 – 29, 1996
Class: 1-390 and Lab: 1-115

Alan Edelman[1] MIT
Robert Schreiber[2] Hewlett Packard
Shang-Hua Teng[3] U. Minnesota

[1]Department of Mathematics and Laboratory for Computer Science. Email: edelman@math.mit.edu, http://theory.lcs.mit.edu/~edelman
[2]HP Labs 3L-5, Hewlett Packard Company, 1501 Page Mill Road , Palo Alto, CA 94304-1126, Email: schreiber@hpl.hp.com
[3]Department of Computer Science. Email: steng@cs.umn.edu, http://www.cs.umn.edu/~steng/

Preface

These lecture notes (under development and constant revision, like the field itself) have been used at MIT in a graduate course first offered by Alan Edelman and Shang-Hua Teng during the spring of 1994 (MIT 18.337, Parallel Scientific Computing). This first class had about forty students from a variety of disciplines which include Applied Mathematics, Computer Science, Mechanical Engineering, Chemical Engineering, Aeronautics and Aerospace, and Applied Physics. Because of the diverse backgrounds of the students, the course, by necessity, was designed to be of interest to engineers, computer scientists, and applied mathematicians.

Our course covers a mixture of material that we feel students should be exposed to. Our primary focus is on modern numerical algorithms for scientific computing, and also on the historical trends in architectures. At the same time, we have always felt that students (and the professors) must suffer through hands-on experience with modern parallel machines. Some students enjoy fighting new machines; others scream and complain. This is the reality of the subject. In 1995, the course was taught again by Alan Edelman with an additional emphasis on the use of portable parallel software tools. The sad truth was that there were not yet enough fully developed tools to be used. The situation is currently improving.

During 1994 and 1995 our students programmed the 128 node Connection Machine CM5. This machine was the 35th most powerful computer in the world in 1994, then the very same machine was the 74th most powerful machine in the spring of 1995. At the time of writing, December 1995, this machine has sunk to position 136. The fastest machine in the world is currently in Japan. In the 1996 course we used the IBM SP-2 and Boston University's SGI machines.

In addition to coauthors Shang-Hua Teng and Robert Schreiber, I would like to thank our numerous students who have written and commented on these notes and have also prepared many of the diagrams. We also thank the students from Minnesota SCIC 8001 and the summer course held at MIT, Summer 6.50s (also taught by Rob Schreiber) for all of their valuable suggestions. These notes will probably evolve into a book which will eventually be coauthored by Rob Schreiber and Shang-Hua Teng. Meanwhile, we are fully aware that the 1996 notes are incomplete, contain mathematical and grammatical errors, and do not cover everything we wish. They are an improvement over the 1995 notes, but not as good as the 1997 notes will be. I view these notes as a basis on which to improve, not as a completed book.

It has been our experience that some students of pure mathematics and theoretical computer science are a bit fearful of programming real parallel machines. Students of engineering and computer science are sometimes intimidated by mathematics. The most successful students understand that computing is not "dirty" and mathematical knowledge is not "scary" or "useless," but both require hard work and maturity to master. The good news is that there are many jobs both in the industrial and academic sectors for experts in the field!

A good course should have a good theme. We try to emphasize the fundamental algorithmic ideas and machine design principles. We have seen computer vendors come and go, but we believe that the mathematical, algorithmic, and numerical ideas discussed in these notes provide a solid foundation that will last for many years.

Contents

III Multipole Methods 69

IV Numerical Linear Algebra 92

V Graph and Geometric Algorithms 144

VI Selected Topics 174

Chapter 1

Introduction

This class strives to study that elusive mix of theory and practice so important for understanding modern scientific computing. We try to cover it all, from the engineering aspects of computer science (parallel architectures, vendors, parallel languages, and tools) to the mathematical understanding of numerical algorithms, and also certain aspects from theoretical computer science.

1.1 A General Look at the Industry

This is a science course at MIT, but somehow it seems appropriate to wonder how special purpose parallel computation really is. Maybe this sort of analysis more properly belongs in the introduction to a business school case study.

Is it correct to conclude that there will never be an enormous market for parallel computers? We are certain that parallel computers will be everywhere. The editorial below may not immediately substantiate my conclusion, but it does illustrate the idea in the famous quote by Shaw: "You see things; and you say, 'Why?' But I dream things that never were; and I say, 'Why not?'

> **Editorial on markets for technology:** We probably all have our favorite examples. How many of us have heard that there will never be a large market for computers. Only "nerds" would have a computer in the living room. I for one have systematically heard that 1) only nerds would communicate by electronic mail, 2) French people would never embrace electronic mail, and 3) lawyers would never accept electronic mail. I suspect that 100 years ago people might have thought that nobody would want a telephone in their living rooms. The surest way to be wrong is to reason that nobody is buying it now hence nobody will ever buy it.

Some books might start with a machine and a programming language. This is a mistake. The machines have been changing and the programming languages are maturing, but still in 1996 the subject is in a state of flux. Many machines are still not as usable as people would want. Quite likely, too many companies have been trying to create their own proprietary products.

> **Editorial on proprietary products:** Look at the fast evolving world wide web. For years we wondered why we were all content to live with `ftp`. One of us used to ask why my bank account balance was unavailable from my workstation. The other now wonders why his telephone is not an integrated part of his workstation. Somehow, just now, the infrastructure and the public's perception is right for the internet to catch on, not only for techies, but for everybody.

However, an interesting phenomenon is developing. Various internet providers are attempting to be the sole owners of the internet. They are learning the hard way that attempts to own the market are doomed to failure. The realists will understand that the web is public, and a common infrastructure benefits all. Attempts to dominate the internet are doomed to failure. Bill Gates, beware!

Parallel computers do not yet have a common infrastructure, but things are improving. Nevertheless parallel machines continue to be more painful to use than serial machines.

Editorial: What do scientists and students with scientific applications want on a parallel machine? People get religious about programming languages, and among the students, even the engineering students, C is growing in importance over Fortran. Nevertheless if we had to pick four things that would make a large number of people very happy we would have to list

- A reliable High Performance Fortran Compiler
- No difficulties using the Message Passing Interface MPI
- A high quality portable common debugging and visualizing mechanism such as Thinking Machines Corporation's Prism product. It is crucial that every machine support the same product.
- A non-proprietary performance monitoring interface whose timings one can trust and allow for predictions. Vendors just do not seem to be able to solve the engineering question of providing the user with the answer to the question of just how long did the program take to execute.

[To be written: discussions about parallel libraries]

Which specific machines did we use in our class? Last year (1995) we used MIT's 128 processor CM-5, and a 12 processor SP-2. This year we expect to use the SP-2 and some Silicon Graphics machines currently located at Boston University. If we are lucky, the MIT lab for Computer Science will receive a donation of an interesting symmetric multiprocessor that we will use in our class.

Most of you may have either seen the 1993 movie or read the 1990 Michael Crichton novel *Jurassic Park*. The 1993 movie shows the CM-5 in the background of the control center and even mentions the Thinking Machines Computer, but you could easily miss it if you do not pay attention. The older novel has the dinosaur theme park controlled by a Cray vector supercomputer. Many in this field believe that the Cray belonged on the dinosaur side of the *Jurassic Park* fence!

In fact, in 1996, both vector supercomputers and proprietary, custom-engineered parallel computers may be on their way to extinction. The investments required to keep pace with evolving hardware technology can't be justified by the size of the market. At Cray, for example, the vector supercomputer (the T90) at 60 gigaflops and 8 gigabytes, is overshadowed by the new parallel machine (the T3E), based on the DEC alpha chip, which offers up to 1.2 teraflops and 4 terabytes. Kendall Square Research designed its machine from the ground up in order to exploit novel ideas about caching and memory architecture: they are now out of business. Tera Computer Company, which has followed the same business model as KSR, will very likely suffer the same fate. And the first big parallel computer company, Thinking Machines Corporation, once located on expensive Cambridge real estate near MIT, has trimmed itself, moved to less-expensive Bedford, and has emerged from bankruptcy protection. Their latest announcement has them teaming up with Sun Microsystems for software for the next generation of machines.

1.2 The Web as a Tool for Tracking Machines

For the latest info on the parallel machine market, the World Wide Web is unmatched. The address `http://www.cs.cmu.edu/~scandal/vendors.html` currently (January 1996) points to Alta Technology. Chen Systems, Convex Computer Corporation, Cray Research, Digital Equipment Corporation, Fujitsu, Hewlett-Packard, Hitachi, IBM, ICE, Intel, Meiko, NEC, Parsytec, Siemens Nixdorf, TERA Computer, and Thinking Machines Corporation.

1.3 Scientific Computing and High Performance Computing

There is no formal definition of *parallel scientific computing* and we do not intend to give one here. However, the goal of parallel scientific computing is very well defined, that is, to find faster solutions to larger and more complex problems. Parallel scientific computing is a highly interdisciplinary area ranging from numerical analysis and algorithm design to programming languages, computer architectures, software, and real applications. The participants include engineers, computer scientists, applied mathematicians, and physicists, and this team is growing.

High-performance parallel computers have become a fundamental tool for computation intensive tasks in real applications. Computer architecture is driven by technology. The advance of very large scale integration (VLSI) makes it possible to develop faster computers that have smaller physical volume. The new technology makes feasible massive parallelism. To sustain increases in computation demands, the industry must embrace multiprocessing in almost all computers, either in the format of tightly coupled multiprocessor supercomputers, or loosely coupled workstation clusters.

Scientific numerical computing plays a critical role in the development of high-performance parallel computers. It provides a class of problems that are so demanding of computational cycles, a class of problems that do not require completely "general purpose" machine, yet are rich in content and in real applications, and a class of problems that give raise challenging scientific and engineering activities ranging from mathematical modeling, algorithmic design to software/hardware/architecture development. Scientific applications were the main motivation for the development of supercomputers in the past decade. Now, parallel machines have begun to find more applications in symbolic computing, large-scale databases, and multimedia environments.

Conversely, parallel computers are the most important platforms for scientific simulation. To perform meaningful simulation within a reasonable time-span computational scientists have to make trade-offs between accuracy, time, and space. The larger memory a machine has, the more powerful it is, the less restriction and simplification we need to impose on the numerical approximation of real problems and their physical domains, and the more accurately we can solve real problems.

1.4 A Natural 3D Problem

The most challenging problems in parallel scientific computing are those in three or higher dimensions. One simple reason is that problems in higher dimensions are of a very large scale.

Parallel scientific computing itself is a three dimensional discipline whose three major dimensions are

$$\begin{aligned} x &= \text{scientific and engineering applications} \\ y &= \text{mathematical algorithms} \end{aligned}$$

$$z = \text{parallel architecture and programming}$$

Our goal in the course is to provide a reliable road map to the projection of our field into the (y, z) plane!

1.5 Components of a Parallel System

The components of a parallel system are not just its hardware, architecture, and network topology. They include its operating system, parallel languages and their compilers, scientific libraries and other software tools.

Most parallel languages are defined by adding parallel extensions to well-established sequential languages, such as C and Fortran. Such extensions allow user to specify various levels of parallelism in an application program, to define and manipulate parallel data structures, and to specify message passing among parallel computing units.

Compilation has become more important for parallel systems. The purpose of a compiler is not just to transform a parallel program in a high-level description into machine-level code. The role of compiler optimization is more important. We will always have discussions about: *Are we developing methods and algorithms for a parallel machine or are we designing parallel machines for algorithm and applications*? Compilers are meant to bridge the gap between algorithm design and machine architectures. Extension of compiler techniques to run time libraries will further reduce users' concern in parallel programming.

Software libraries are important tools in the use of computers. The purpose of libraries is to enhance the productivity by providing preprogrammed functions and procedures. Software libraries provide even higher level support to programmers than high-level languages. Parallel scientific libraries embody expert knowledge of computer architectures, compilers, operating systems, numerical analysis, parallel data structures, and algorithms. They systematically choose a set of basic functions and parallel data structures, and provide highly optimized routines for these functions that carefully consider the issues of data allocation, data motion, load balancing, and numerical stability. Hence scientists and engineers can spend more time and be more focused on developing efficient computational methods for their problems. Another goal of scientific libraries is to make parallel programs portable from one parallel machine platform to another. Because of the lack, until very recently, of non-proprietary parallel programming standards, the development of portable parallel libraries has lagged far behind the need for them. There is good evidence now, however, that scientific libraries will be made more powerful in the future and will include more functions for applications to provide a better interface to real applications.

Due to the generality of scientific libraries, their functions may be more complex than needed for a particular application. Hence, they are less efficient than the best codes that take advantage of the special structure of the problem. So a programmer needs to learn how to use scientific libraries. A pragmatic suggestion is to use functions available in the scientific library to develop the first prototype and then to iteratively find the bottleneck in the program and improve the efficiency.

1.6 Scientific Algorithms

The multidisciplinary aspect of scientific computing is clearly visible in algorithms for scientific problems. A scientific algorithm can be either sequential or parallel. In general, algorithms in scientific software can be classified as graph algorithms, geometric algorithms, and numerical algorithms, and most scientific software calls on algorithms of all three types. Scientific software also makes use of advanced data structures and up-to-date user interfaces and graphics.

1.7 Applications

Parallel (numerical) methods have been used to solve problems in virtually all areas of science and engineering:

- Structural analysis and mechanical engineering.
- Computational chemistry and material science
- Computational particle physics
- Computational fluid and plasma dynamics, air and space vehicle design, and space science.
- Environment and Earth science: climate modeling, glaciology, weather forecasting, an ocean modeling.
- Structure biology and medical imaging, signal processing, drug design
- ECAD and circuit validation
- Economics and finance
- Optimization, searching and databases
- Oil and gas exploration

1.8 History, State-of-Art, and Perspective

The supercomputer industry started when Cray Research developed the Cray-1, in 1975. The massively parallel processing (MPP) industry emerged with the introduction of the CM-2 by Thinking Machines in 1985. The industry is represented by several companies including Cray Research (the T90 and the T3E), Intel Supercomputer System Division (the Paragon), Thinking Machines Corporation (the CM-5 and the Global), IBM (the SP-2), HP (the Convex Exemplar), MasPar (the MP-2), Tera (the MTA, which is also the subway system in Boston), and Silicon Graphics (the PowerChallenge). The MPP industry has received substantial support from the U.S. government. The *US High Performance Computing and Communication* program was designed to explore technologies for building and using such machines. Most MPP installations are at government research centers (e.g., NASA Ames and Los Alamos) and universities and supercomputer centers (NCSA, Pittsburgh, MSI, etc).

Achieving high performance for numerical computing for scientific applications are the main goals. Indeed, most of the benchmarks are numeric oriented such as dense linear algebra computations, as represented by the heated race for the top of Linpack benchmarks between Intel and Thinking Machines.

High performance parallel computers have begun to play an important role in science and engineering development. At the same time, the vendors of parallel supercomputers have engaged in a brand of hype unusual in science and engineering. Now that the cold war has ended, the supercomputing market in major universities and government centers is close to being saturated, at least for the moment. (Nevertheless, the U.S. Department of Energy has, this past year, purchased a massively parallel supercomputer that far exceeds the capacity of any previously built.) The parallel computing industry is evolving. There are more and more "real" demands from commercial companies. Hence the cost/efficiency trade-off is more relevant. More friendly programming environments are desired. Other non-numeric applications such as in large databases, optimization, business and finance are in greater demand. And of course the parallel computing industry is more competitive. Workstation clusters bring new challenges to tightly coupled MPP. The parallel computing field and industry are maturing. The application domain of parallel computing is expanding.

1.9 Parallel Computing: An Example

Here is an example of a problem that one can easily imagine being performed in parallel:

A rudimentary parallel problem: Compute

$$x_1 + x_2 + \ldots + x_P,$$

where x_i is a floating point number and for purposes of exposition, let us assume P is a power of 2.

Obviously, the sum can be computed in $\log P$ (base 2 is understood) steps, simply by adding neighboring pairs recursively. (The algorithm has also been called *pairwise summation* and *cascade summation*). The data flow in this computation can be thought of as a binary tree.

(Illustrate on tree.)

Nodes represent values, either input or the result of a computation. Edges communicate values from their definition to their uses.

This is an example of what is often known as a *reduce* operation. We can replace the addition operation with any associative operator to generalize. (Actually, floating point addition is not associative, leading to interesting numerical questions about the best order in which to add numbers. This is studied by Higham [46, See p. 788].)

There are so many questions though. How would you write a program to do this on P processors? Is it likely that you would want to have such a program on P processors? How would the data (the x_i) get to these processors in the first place? Would they be stored there or result from some other computation? It is far more likely that you would have $10000P$ numbers on P processors to add.

A correct parallel program is often not an efficient parallel program. A flaw in a parallel program that causes it to get the right answer *slowly* is known as a performance bug. Many beginners will write a working parallel program, obtain poor performance, and prematurely conclude that either parallelism is a bad idea, or that it is the machine that they are using which is slow.

What are the sources of performance bugs? We illustrate some of them with this little, admittedly contrived example. For this example, imagine four processors, numbered zero through three, each with its own private memory, and able to send and receive message to/from the others. As a simple approximation, assume that the time consumed by a message of size n words is $A + Bn$.

Four Bad Parallel Algorithms for Computing $1 + 2 + 3 + \cdots + 10^6$ on Four Processors

1. Load imbalance. One processor has too much to do, and the others sit around waiting for it. For our problem, we could eliminate all the communication by having processor zero do all the work; but we won't see any parallel speedup with this method!

2. Excessive communication.

   Processor zero has all the data which is divided into four equally sized parts each with a quarter of a million numbers. It ships three of the four parts to each of the other processors for proper load balancing. The results are added up on each processor, and then processor 0 adds the final four numbers together.

True, there is a tiny bit of load imbalance here, since processor zero does those few last additions. But that is nothing compared with the cost it incurs in shipping out the data to the other processors. In order to get that data out onto the network, it incurs a large cost that does not drop with the addition of more processors. (In fact, since the number of messages it sends grows like the number of processors, the time spent in the initial communication will actually increase.)

3. A sequential bottleneck. Let's assume the data are initially spread out among the processors; processor zero has the numbers 1 through $250,000$, etc. Assume that the owner of i will add it to the running total. So there will be no load imbalance. But assume, further, that we are constrained to add the numbers in their original order! (Sounds silly when adding, but other algorithms require exactly such a constraint.) Thus, processor one may not begin its work until it receives the sum $0 + 1 + \cdots + 250,000$ from processor zero!

   We are thus requiring a sequential computation: our binary summation tree is maximally unbalanced, and has height $1,000,000$. It is always useful to know the critical path – the length of the longest path in the dataflow graph – of the computation being parallelized. If it is excessive, change the algorithm!

4. A scheduling problem. For this kind of performance bug, we get a better illustration by changing the problem. Let there be a 1000×1000 matrix, distributed to the four processors by columns, so that processor zero owns all of columns 1 through 250. The job is to add the columns, producing a vector of length 1000. The constraint, which makes this problem interesting, is that within each row we are required to do the addition left-to-right! So the computation in each matrix row is sequential, and the parallelism comes from the fact that there are 1000 independent row sums to be computed.

   The bad algorithm has processor zero do all of its row sums, then send them all in a message of 1000 words to processor one, etc. We have chosen to defer sending the sum of the first rows until we compute the sum of the last rows, thereby sequentializing an otherwise parallel task. The right idea is to schedule high-priority tasks (the sending of sums to the next processor) sooner than lower priority tasks. The fact of the latency term A in message cost makes the optimum scheduling here tricky – single-word messages may not be ideal either.

1.10 Exercises

1. Compute the sum of 1 through $1,000,000$ using HPF. This amounts to a "hello world" program on whichever machine you are using. We are not currently aware of any free distributions of HPF for workstations, so your instructor will have to suggest a computer to use.

2. Download MPI to your machine and compute the sum of 1 through $1,000,000$ using C or Fortran with MPI. A number of MPI implementations may be found at `http://www.mcs.anl.gov/Projects/mpi/implementations.html` The easy to follow ten step quick start on `http://www.mcs.anl.gov/mpi/mpiinstall/node1.html#Node1` worked very easily on the MIT mathematics department's SUN network.

3. In HPF, generate $1,000,000$ real random numbers and sort them. (Use `RANDOM_NUMBER` and `GRADE_UP`.
4. (Extra Credit) Do the same in C or Fortran with MPI.
5. Set up the excessive communication situation described as the second bad parallel algorithm. Place the numbers 1 through one million in a vector on one processor. Using four processors see how quickly you can get the sum of the numbers 1 through one million.

I Mathematical and Computer Science Foundations

Modeling and solving complex scientific problems requires a large class of mathematical machinery and a proper understanding of real machines. In this part of the book, we will review some basic notions in mathematics and computer science.

Chapter 2

Linear Algebra

2.1 Linear Algebra

Linear algebraists often like to believe that every scientific application, once discretized, becomes a linear algebra problem. Though many applications have no linear algebra at all in them, there is no question that linear algebra at least covers a very large fraction of applications. Here we intend to review the most important ideas of the field that are needed to follow the lecture notes. Meanwhile, a good text such as the elementary treatment by Strang or the numerical coverage by Golub and van Loan [40] should be consulted.

Experience has tuaght me that the first idea that students forget is the concept of eigenvalues. An eigenvalue is simply a number λ such that $Ax = \lambda x$. If a matrix has a full set of eigenvectors , this can be expressed with the notation $AX = X\Lambda$, where the columns of X are the eigenvectors, and the diagonal matrix Λ contains the eigenvalues.

Once linear algebra was taught more abstractly than it usually is now. In a classical course, a matrix A is a linear operator on a vector space. Now it is usually an $n \times n$ array of numbers. For the largest applications, the old fashioned way of thinking may be more the right way. It is probably best to think of a matrix A as a black-box whose input x (call it a vector if you like) gets transformed into the output Ax. The student who understands that polynomials form a vector space and differentiation is a perfectly legitimate linear operator (with determinant 0) understands the concept.

For more medium sized problems, the n by n array point of view makes sense, and then this black box may be thought of as "matrix–vector" muliplication in the usual component sense.

Chapter 3

Function Expansions

3.1 Function Representation and Expansions

How should we represent functions? Much of the history of mathematics or at least what we now call analysis might be traced to this very question.

We hardly ever think of a power series as a definition of a function, but it is certainly true that $\sum_{k=0}^{\infty} z^k/k!$ defines the exponential function. Even $\sum_{k=0}^{\infty} z^k$ defines $1/(1-z)$, and although the power series has no meaning for $|z| > 1$, we somehow feel that the power series is somehow good enough to capture the function $f(z) = 1/(1-z)$ everywhere, even though technically the series does not converge for $|z| > 1$.

So what is a power series? From the calculus point of view, it is always presented as a representation of a function which lets you plug in a number z, and if z is within this radius of convergence the function seems well defined. Another point of view is that a power series is a linear combination of the "basis" functions $f(z) = z^k$, $k = 0, \ldots, \infty$.

There is yet another point of view which is not considered nearly as often. Any function can be interpolated with an nth degree polynomial given values at $n+1$ points. (I always like to remember the $n = 1$ story that "two points determine a line.") If those $n+1$ points all coalesce into the same point, the polynomial is the leading segment of the Taylor series. Therefore if we can imagine interpolating a function using "infinitely" many points at the origin, we get the Taylor or power series. A Taylor series about another point z_c has the form

$$f(z) = \sum_{k=0}^{\infty} a_k (z - z_c)^k$$

Another basis for functions that is natural may be derived by taking one generating function and using all of its derivatives as the basis. For example if we start with $f(z) = 1/z$, then we obtain series expansions of the form $\sum_{k=0}^{\infty} a_k/z^k$. This is called a *multipole expansion* centered at $z = 0$. More generally, if our generator is $1/(z - z_c)$, then we have the general form of the multipole expansion of a function $f(z)$

$$f(z) = \sum_{k=1}^{\infty} a_k \frac{1}{(z - z_c)^k}.$$

If the generator is $\log(z - z_c)$ then of course the multipole expansion has one more term:

$$f(z) = a_0 \log(z - z_0) + \sum_{k=1}^{\infty} a_k \frac{1}{(z - z_c)^k}.$$

Multipole expansions converge outside of a disk in the complex plane. By contrast, the more familiar Taylor series converge within a disk.

We can take any function $f(z)$ and write it as a Taylor series about an arbitrary point z_c. Forgetting for a moment what the pure mathematicians might think, let us agree that at least psychologically, all such Taylor series are just multiple ways of representing the same function. Also, the same $f(z)$ amy be written as a multipole expansion; each different center z_c gives a different representation with different covergence properties.

If

$$f(z) = {}_{k=1}^{\infty}\ a_k \frac{1}{(z - z_c)^k} = {}_{k=1}^{\infty}\ a'_k \frac{1}{(z - z'_c)^k} \tag{3.1}$$

then the linear operator that takes the a_k to the a'_k is known as a shift. Similarly if

$$f(z) = {}_{k=0}^{\infty}\ a_k (z - z_c)^k = {}_{k=0}^{\infty}\ a'_k (z - z'_c)^k \tag{3.2}$$

this operator is also known as a shift. Finally, if

$$f(z) = {}_{k=1}^{\infty}\ a_k \frac{1}{(z - z_c)^k} = {}_{k=0}^{\infty}\ a'_k (z - z'_c)^k \tag{3.3}$$

then the operator that converts the a_k to an a'_k is known as a flip. We will see that multipole expansions serve the purpose of broadcasting the news about a body to far away locations efficiently. On the other hand, Taylor expansions serve the purpose of locally computing all the needed information. The shifts allow the computations to occur on common ground.

3.2 A close look at shifts and flips

Given fixed values of z_c and z'_c, the equations (3.1), (3.2), and (3.3) define linear operators from the infinite a vector to the infinite a' vector.

In a moment we will work out an explicit form for the operators, but mathematics students should be able to see the linearity from a slightly abstract viewpoint. Ignoring issues of infinity and convergence, the operator from a and a' vectors to formal functions is clearly linear. Since the zero function has only one formal representation, the map from a to a' must be linear.

[work out explicit formulas and remove from the particle section]

3.3 Harmonic expansions in 3-dimensions

[This is a specialized optional topic and is usually omitted from the course]

3.4 Exercises

1. For a general derivative series work out the shift formula. Are you surprised?

2. Work out the flip formula from a general derivative series to a general integral series.

3. Work out the shift formula from a general integral series to another integral series.

4. Numerically shift and flip the field due to one charge. Compare the absolute and relative errors with that of the true shifted or flipped field. Difficult: Derive a theoretical estimate for the difference and compare with what is seen.

Chapter 4

Graph Theory

Ask a student from an engineering discipline what a *graph* is. The student will probably right away say something about the plot of a function with axes and a curve that represents the graph of a function.

If you ask a computer scientist or a mathematician, you find out that a *graph* is a collection of vertices and edges that indicate connections between the vertices. Mathematically, a graph $G = (V, E)$ consists a set V of vertices that are connected by a set E of edges. A typical graph is given in Figure **??**.

A graph can be either directed or undirected. In an *undirected graph*, E is a set of unordered pairs of vertices. In contrast, in a *directed graph* $G = (V, E)$, E is a set of ordered pairs of vertices, where the order indicates the direction of an edge.

Trees, planar graphs, an example of a non-planar graph.

Graphs may be represented in many ways: edge lists; it's Laplacian (see Chapter ??); adjacent matrices, node edge incident matrices.

Basic concepts: degrees, paths, cycles, independent sets, colorings, connected components,

For many applications, we can associate weights with edges and vertices to define a weighted graph.

Chapter 5

Elementary Geometry

Chapter 6

Probability Theory

Chapter 7

Computer Architecture

Anybody who uses a computer ought to have an understanding of basic architectual components such as memory, registers, caches, and buses.

The actual data that a computer works with is stored as binary digits (0's and 1's) on memory chips. This collection of chips is often denoted *main memory* (or DRAM in the PC ads). The minimum number of bits (binary digits) that can be accessed by one request to memory is known as a *word*. Word sizes are almost always one *byte* (meaning eight bits) long. Single precision numbers are usually four words long, while double precision are eight words long. Computers access multiple words simultaneously.

If M is the total number of words available on a machine, then a unique word may be specified by an address from 0 to M-1.

If words are always eight bits wide, what is all this media fuss about 32 bit technology, or 64 bit technology? These numbers (32 or 64) represent $\log_2(M)$, i.e., with 64 bit technology, there are potentially $M = 2^{64}$ words available on the machine. The truth is that there are often fewer words actually available, but these many are at least potentially available. In a given United States area code, we might say we have seven digit "technology", but not all 10^7 numbers have been given away at any one time. It is always good to leave room for more. Furthermore, some memory addresses may not point to words in memory at all, but rather to other components where you may want to send bits.

To actually work with data, a processor must load it into its own *registers*. Registers are special storage areas on a processor that may be quickly accessed.

Words travel from memory to registers via *buses*. A bus is a piece of hardware that allows the various components of a computer to talk to each other.

It was well understood in the 1960s that the time taken to load data from memory into registers slowed down machine execution time. Most memory simply can not release the data fast enough for the processor to work with the data. Therefore a device designed to accelerate the average transfer time called a *cache* was invited.

Caches are small amounts of fast memory which contains copies of a small subset of the data in main memory. When a processor wants to load data from memory, the cache is checked to see if the data is there. If so we get a fast load which is called a *cache hit*. If not, we get a slow load called a *cache miss*. When a cache miss occurs, a contiguous block of data containing this word of data is loaded into cache anticipating that having the data and its neighbors in cache is likely to increase the chance of a cache hit the next time.

Therefore, it is of utmost important to design algorithms that exhibit this sort of locality whenever possible.

II Parallelism

Chapter 8

Parallel Machines

A parallel computer is a connected configuration of processors and memories. The choice space available to a computer architect includes the network topology, the node processor, the address-space organization, and the memory structure. These choices are based on the parallel computation model, the current technology, and marketing decisions.

8.1 Parallel Computer Architecture

As we write this in February 1996, we fear that what we are about to write may well be outdated by next year. No doubt this is the most ephemeral section of our notes. The rapid pace of change is exciting to watch, but frustrating to write down. No matter what the pace of change, it is impossible to make intelligent decisions about parallel computers right now without some knowledge of their architecture. For more advanced treatment of computer architecture we recommend Kai Hwang's *Advanced Computer Architecture* and the soon to be completed *Parallel Computer Architecture* by Gupta, Singh, and Culler.

One may gauge what architectures are important today by the Top500 Supercomputer list published by Dongarra, Meuer, and Strohmaier. The secret is to learn to read between the lines. There are five kinds of machines on **The November 10, 1995 Top 500 list**:

- Distributed Memory Multicomputers
- Central Memory Symmetric Multiprocessors (SMPs)
- Networks (Arrays) of SMPs
- Vector Supercomputers
- Single Instruction Multiple Data (SIMD) Machines

It seems fair to say the last two (vector machines and SIMD machines) are on the way out (more about this later.)

How can one simplify (and maybe grossly oversimplify) the current situation? Perhaps by pointing out that the world's fastest machines are all distributed memory multicomputers, but the next batch of machines are symmetric multiprocessors and arrays of SMPs. SMPs are very quickly gaining in popularity, and it is very likely that an array of SMPs will be in the top ten very soon. Perhaps it will be helpful to the reader to list some of the most important machines first sorted by type, and then by highest rank in the top 500 list.

Machine	Top 500 First Rank
Distributed Memory	
Fujitsu NWT	1
Intel XP/S	2
Cray T3D	4
Fujitsu VPP500	5
IBM SP2	6
TMC CM-5	10
Hitachi S-3800	27
Intel Delta	59
Parsytech GC Power Plus	128
IBM 9076	233
Meiko CS-2	221
KSR2-80	235

Machine	Top 500 First Rank
SMP Arrays	
SGI Power Challenge Array	30
SMP	
Digital AlphaServer 8400	227
SGI Power Challenge	240
Convex SPP1200	298
Vector Machines	
NEC SX	9
Cray YMP	60
Fujitsu VP2600	273
SIMD Machines	
TMC CM-200	119

The trend is clear to anyone who looks at the list. Distributed memory machines capture the high end of the supercomputing market; the next level of supercomputers are arrays of SMPs.

Distributed Memory Multicomputers:

Remembering that a computer is a processor and memory, really a processor with cache and memory, it makes sense to call a set of such "computers" linked by a network a *multicomputer*. Figure 8.1 shows 1) a basic computer which is just a processor and memory and also 2) a fancier computer where the processor has cache, and there is auxiliary disk memory. To the right, we picture 3) a three processor multicomputer. The line on the right is meant to indicate the network.

These machines are sometimes called distributed memory multiprocessors. We can further distinguish between DMM's based on how each processor addresses memory. We call this the private/shared memory issue:

> **Private versus shared memory on distributed memory machines:** It is easy to see that the simplest architecture allows each processor to address only its own memory. When a processor wants to read data from another processor's memory, the owning processor must send the data over the network to the processor that wants it. Such machines are said to have *private memory*. A close analog is that in my office, I can easily find information that is located on my desk (my memory) but I have to make a direct request via my telephone (that is, I must dial a phone number) to find information that is not in my office. And I have to hope that the other person is in his or her office, waiting for the phone to ring. In other words, I need the cooperation of the other active party (the person or processor) to be able to read or write information located in another place (the office or memory).
>
> The alternative is a machine in which every processor can directly address every memory. Such a machine is said to have a *shared address space*, and sometimes informally *shared memory*, though the latter terminology is misleading as it may easily be confused with machines where the memory is physically shared. On a shared address space machine, each processor can load data from or store data into the memory of any processor, without the active cooperation of that processor. When a processor requests memory that is not local to it, a piece of hardware intervenes and fetches the data over the network. Returning to the office analogy, it would be as if I asked to view some information that happened to not be in my office, and some special assistant actually

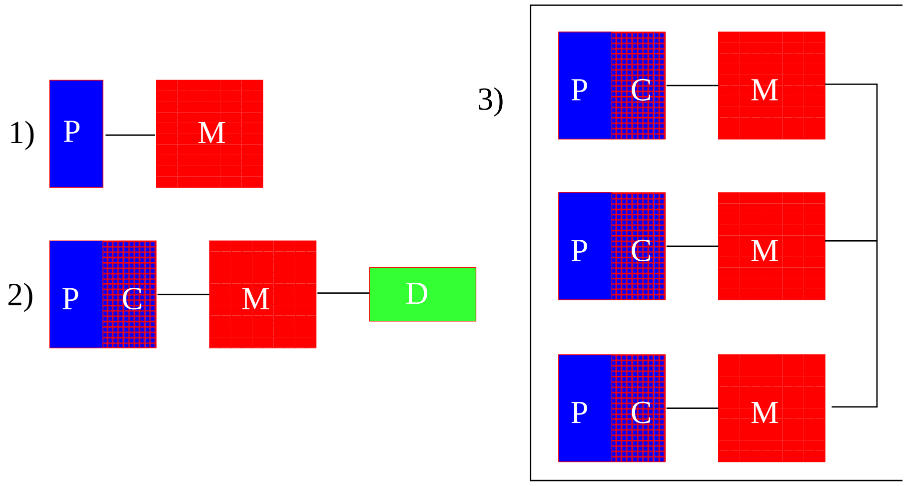


Figure 8.1: 1) A computer 2) A fancier computer 3)A multicomputer

dialed the phone number for me without my even knowing about it, and got a hold of the special assistant in the other office, (and these special assistants never leave to take a coffee break or do other work) who provided the information.

Most distributed memory machines have private addressing. One notable exception is the Cray T3D and the Fujitsu VPP500 which have shared physical addresses.

Central Memory Symmetric Multiprocessors (SMPs):

Notice that we have already used the word "shared" to refer to the shared address space possible in in a distributed memory computer. Sometimes the memory hardware in a machine does not obviously belong to any one processor. We then say the memory is *central*, though some authors may use the word "shared." Therefore, for us, the central/distributed distinction is one of system architecture, while the shared/private distinction mentioned already in the distributed context refers to addressing.

> **"Central" memory contrasted with distributed memory:** We will view the physical memory architecture as distributed if the memory is packaged with the processors in such a way that some parts of the memory are substantially "farther" (in the sense of lower bandwidth or greater access latency) from a processor than other parts. If all the memory is nearly equally expensive to access, the system has central memory. The vector supercomputers are genuine central memory machines. A network of workstations has distributed memory.

Microprocessor machines known as *symmetric multiprocessors* are becoming typical now as mid-sized compute servers; this seems certain to continue to be an important machine design. On these machines each processor has its own cache while the main memory is central. There is no one "front-end" or "master" processor, so that every processor looks like every other processor. This is the "symmetry." The microprocessors themselves have caches built right onto the chips, so these caches act like distributed, low-latency, high-bandwidth memories, giving the system many of the important performance characteristics of distributed memory. Therefore if one insists on being

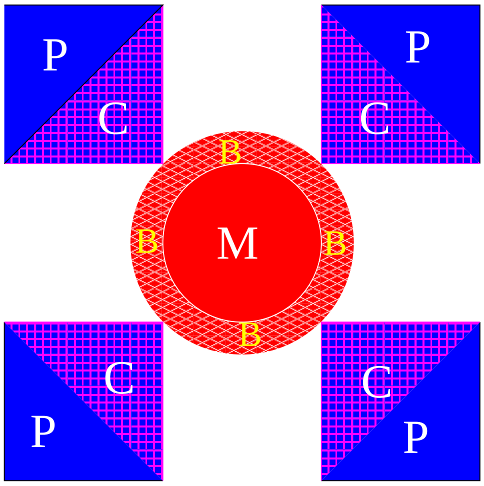


Figure 8.2: A four processor SMP (B denotes the bus between the central memory and the processor's cache

precise, it is not all of the memory that is central, merely the main memory. Such systems are said to have non-uniform memory access (NUMA).

A big research issue for shared memory machines is the cache coherence problem. All fast processors today have caches. Suppose the cache can contain a copy of any memory location in the machine. Since the caches are distributed, it is possible that P2 can overwrite the value of x in P2's own cache and main memory, while P1 might not see this new updated value if P1 only looks at its own cache. Coherent caching means that when the write to x occurs, any cached copy of x will be tracked down by the hardware and invalidated – *i.e.* the copies are thrown out of their caches. Any read of x that occurs later will have to go back to its home memory location to get its new value. Maintenance of cache coherence is expensive for scalable shared memory machines. Today, only the HP Convex machine has scalable, cache coherent, shared memory. Other vendors of scalable, shared memory systems (Kendall Square Research, Evans and Sutherland, BBN) have gone out of the business. Another, Cray, makes a machine (the Cray T3E) in which the caches can only keep copies of local memory locations.

SMPs are often thought of as not scalable (performance peaks at a fairly small number of processors), because as you add processors, you quickly saturate the bus connecting the memory to the processors.

SIMD machines:

In the late 1980's, there were debates over SIMD versus MIMD. (Either pronounced as SIM-dee/MIM-dee or by reading the letters es-eye-em-dee/em-eye-em-dee.) These two acronyms coined by Flynn in his classification of machines refer to **Single Instruction Multiple Data** and **Multiple Instruction Multiple Data**. The second two letters, "MD" for multiple data, refer to the ability to work on more than one operand at a time. The "SI" or "MI" refer to the ability of a processor to issue instructions of its own. Most current machines are MIMD machines. They are built from microprocessors that are designed to issue instructions on their own. One might say that each processor has a brain of its own and can proceed to compute anything it likes independent of what the other processors are doing. On a SIMD machine, every processor is executing the same instruction, an add say, but it is executing on different data.

SIMD machines need not be as rigid as they sound. For example, each processor had the ability to not store the result of an operation. This was called em context. If the contest was false, the result was not stored, and the processor appeared to not execute that instruction. Also the CM-2 had the ability to do indirect addressing, meaning that the physical address used by a processor to load a value for an add, say, need not be constant over the processors.

The most important SIMD machines were the Connection Machines 1 and 2 produced by Thinking Machines Corporation, and the MasPar MP-1 and 2. The SIMD market received a serious blow in 1992, when TMC announced that the CM-5 would be a MIMD machine.

Now the debates are over. MIMD has won. The prevailing theory is that because of the tremendous investment by the personal computer industry in commodity microprocessors, it will be impossible to stay on the same steep curve of improving performance using any other processor technology. *"No one will survive the attack of the killer micros!"* said Eugene Brooks of the Lawrence Livermore National Lab. He was right. The supercomputing market does not seem to be large enough to allow vendors to build their own custom processors. And it is not realistic or profitable to build an SIMD machine out of these microprocessors. Furthermore, MIMD is more flexible than SIMD; there seem to be no big enough market niches to support even a single significant vendor of SIMD machines.

A close look at the SIMD argument:
In some respects, SIMD machines are faster from the communications viewpoint. They can communicate with minimal latency and very high bandwidth because the processors are always in synch. The Maspar was able to do a circular shift of a distributed array, or a broadcast, in less time than it took to do a floating point addition. So far as we are aware, no MIMD machine in 1996 has a latency as small as the 24 μsec overhead required for one hop in the 1988 CM-2 or the 8 μsec latency on the Maspar MP-2.

Admitting that certain applications are more suited to SIMD than others, we were among many who thought that SIMD machines ought to be cheaper to produce in that one need not devote so much chip real estate to the ability to issue instructions. One would not have to replicate the program in every machine's memory. And communication would be more efficient in SIMD machines. Pushing this theory, the potentially fastest machines (measured in terms of raw performance if not total flexibility) should be SIMD machines. In its day, the MP-2 was the world's most cost-effective machine, as measured by the NAS Parallel Benchmarks. These advantages, however, do not seem to have been enough to overcome the relentless, amazing, and wonderful performance gains of the "killer micros".

Continuing with the Flynn classification (for historical purposes) **Single Instruction Single Data** or SISD denotes the sequential (or Von Neumann) machines that are on most of our desktops and in most of our living rooms. (Though most architectures show some amount of parallelism at some level or another.) Finally, there is **Multiple Instruction Single Data** or MISD, a class which seems to be without any extant member although some have tried to fit systolic arrays into this ill-fitting suit.

There have also been hybrids; the PASM Project (at Purdue University) has investigated the problem of running MIMD applications on SIMD hardware! There is, of course, some performance penalty.

Vector Supercomputers:

A vector computer today is a central, shared memory MIMD machine in which every processor has some pipelined arithmetic units and has vector instructions in its repertoire. A vector instruction is something like "add the 64 elements of vector register 1 to the 64 elements of vector register 2", or "load the 64 elements of vector register 1 from the 64 memory locations at addresses $x, x+10, x+20, \ldots, x+630$." Vector instructions have two advantages: fewer instructions fetched, decoded, and issued (since one instruction accomplishes a lot of computation), and predictable memory accesses that can be optimized for high memory bandwidth. Clearly, a single vector processor, because it performs identical operations in a vector instruction, has some features in common with SIMD machines. If the vector registers have p words each, then a vector processor may be viewed as an SIMD machine with shared, central memory, having p processors.

Hardware Technologies and Supercomputing:

Vector supercomputes have very fancy integrated circuit technology (bipolar ECL logic, fast but power hungry) in the processor and the memory, giving very high performance compared with other processor technologies; however, that gap has now eroded to the point that for most applications, fast microprocessors are within a factor of two in performance. Vector supercomputer processors are expensive and require unusual cooling technologies. Machines built of gallium arsenide, or using Josephson junction technology have also been tried, and none has been able to compete successfully with the silicon, CMOS (complementary, metal-oxide semiconductor) technology used in the PC and workstation microprocessors. Thus, from 1975 through the late 1980s, supercomputers were machines that derived their speed from uniprocessor performance, gained through the use of special hardware technologies; now supercomputer technology is the same as PC technology, and parallelism has become the route to performance.

SMP arrays, NOWs, Heterogeneous Clusters:

Whenever anybody has a collection of machines, it is always natural to try to hook them up together. Therefore any arbitrary collection of computers can become one big distributed computer. When all the nodes are the same, we say that we have a homogeneous arrangement. It has recently become popular to form high speed connections between SMPs, known as *SMP arrays*. Any collection of workstations is likely to be networked together: this is the cheapest way for many of us to work on parallel machines given that the networks of workstations already exist where most of us work. Sometimes the nodes have different architectures creating a heterogeneous situation. Under these circumstances, it is sometimes necessary to worry about the explicit format of data as it is passed from machine to machine.

8.1.1 More on private versus shared addressing

Both forms of addressing lead to difficulties for the programmer. In a shared address system, the programmer must insure that any two processors that access the same memory location do so in the correct order: for example, processor one should not load a value from location N until processor zero has stored the appropriate value there (this is called a "true" or "flow" dependence); in another situation, it may be necessary that processor one not store a new value into location N before processor zero loads the old value (this is an "anti" dependence); finally, if multiple processors write to location N, its final value is determined by the last writer, so the order in which they write is significant (this is called "output" dependence). The fourth possibility, a load followed by another load, is called an "input" dependence, and can generally be ignored. Thus, the programmer can get incorrect code do to "data races". Also, performance bugs due to too many accesses to the same location (the memory bank that holds a given location becomes the sequential bottleneck) are common. [1]

The big problem created by private memory is that the programmer has to distribute the data. "Where's the matrix?" becomes a key issue in building a LINPACK style library for private memory machines. And communication cost, whenever there is NUMA, is also a critical issue. It has been said that the three most important issues in parallel algorithms are "locality, locality, and locality".[2]

One factor that complicates the discussion is that a layer of software, at the operating system level or just above it, can provide virtual shared addressing on a private address machine by using interrupts to get the help of an owning processor when a remote processor wants to load or store data to its memory. A different piece of software can also segregate the shared address space of a machine into chunks, one per processor, and confine all loads and stores by a processor to its own chunk, while using private address space mechanisms like message passing to access data in other chunks. (As you can imagine, hybrid machines have been built, with some amount of shared and private memory.)

8.1.2 Programming Model

The programming model used may seem to be natural for one style of machine; data parallel programming seems to be a SIMD shared memory style, and message passing seems to favor distributed memory MIMD.

Nevertheless, it is quite feasible to implement data parallelism on distributed memory MIMD machines. For example, on the Thinking Machines CM-5, a user can program in CM-Fortran an array data parallel language, or program in node programs such as C and Fortran with message passing system calls, in the style of MIMD computing. We will discuss the pros and cons of SIMD and MIMD models in the next section when we discuss parallel languages and programming models.

8.1.3 Machine Topology

The two things processors need to do in parallel machines that they do not do when all alone are communication (with other processors) and coordination (again with other processors). Communication is obviously needed: one computes a number that the other requires, for example. Coordination is important for sharing a common pool of resources, whether they are hardware units, files, or a pool of work to be performed. The usual mechanisms for coordination, moreover, involve communication.

Parallel machines differ in their underlying hardware for supporting message passing and data routing.

In a shared memory parallel machine, communication between processors is achieved by access to common memory locations. Access to the common memory is supported by a switch network that connects the processors and memory modules. The set of proposed switch network for shared parallel machines includes crossbars and multistage networks such as the butterfly network. One can also connect processors and memory modules by a bus, and this is done when the number of processors is limited to ten or twenty.

[1] It is an important problem of the "PRAM" model used in the theory of parallel algorithms that it does not capture this kind of performance bug, and also does not account for communication in NUMA machines.

[2] For those too young to have suffered through real estate transactions, the old adage in that business is that the three most important factors in determining the value of a property are "location, location, and location".

An interconnection network is normally used to connect the nodes of a multicomputer as well. Again, the the network topology varies from one machine to another. Due to technical limitations, most commercially used topologies are of small node degree. Commonly used network topologies include (not exclusively) linear arrays, ring, hierarchical rings, two or three dimension grids or tori, hypercubes, fat trees.

The performance and scalability of a parallel machine in turn depend on the network topology. For example, a two dimensional grid of p nodes has diameter $\sqrt{p}$, on the other hand, the diameter of a hypercube or fat tree of p nodes is $\log p$. This implies that the number of physical steps to send a message from a processor to its most distant processor is $\sqrt{p}$ and $\log p$, respectively, for 2D grid and hypercube of p processors. The node degree of a 2D grid is 4, while the degree of a hypercube is $\log p$. Another important criterion for the performance of a network topology is its *bisection bandwidth*, which is the minimum communication capacity of a set of links whose removal partitions the network into two equal halves. Assuming unit capacity of each direct link, a 2D and 3D grid of p nodes has bisection bandwidth $\sqrt{p}$ and $p^{2/3}$ respectively, while a hypercube of p nodes has bisection bandwidth $\Theta(p/\log p)$. (See FTL page 394)

There is an obvious cost / performance trade-off to make in choosing machine topology. A hypercube is much more expensive to build than a two dimensional grid of the same size. An important study done by Bill Dally at Caltech showed that for randomly generated message traffic, a grid could perform better and be cheaper to build. Dally assumed that the number of data signals per processor was fixed, and could be organized into either four “wide” channels in a grid topology or $\log n$ “narrow” channels (in the first hypercubes, the data channels were bit-serial) in a hypercube. The grid won, because too the average utilization of the hypercube channels was too low: the wires, probably the most critical resource in the parallel machine, were sitting idle. Furthermore, the work on routing technology at Caltech and elsewhere in the mid 80’s resulted in a family of hardware routers that delivered messages with very low latency even though the length of the path involved many “hops” through the machines. For the earliest multicomputers used “store and forward” networks, in which a message sent from A through B to C was copied into and out of the memory of the intermediate node B (and any others on the path): this causes very large latencies that grew in proportion to the number of hops. Later routers, including those used in todays networks, have a “virtual circuit” capability that avoids this copying and results in small latencies.

Does topology make any real difference to the performance of parallel machines in practice? Some may say “yes” and some may say “no”. Due to the small size (less than 512 nodes) of most parallel machine configurations and large software overhead, it is often hard to measure the performance of interconnection topologies at the user level.

8.1.4 Homogeneous and heterogeneous machines

Another example of cost / performance trade-off is the choice between tightly coupled parallel machines and workstation clusters, workstations that are connected by fast switches or ATMs. The networking technology enables us to connect heterogeneous machines (including supercomputers) together for better utilization. Workstation clusters may have better cost/efficient trade-offs and are becoming a big market challenger to “main-frame” supercomputers.

A parallel machine can be *homogeneous* or *heterogeneous*. A homogeneous parallel machine uses identical node processors. Almost all tightly coupled supercomputers are homogeneous. Workstation clusters may often be heterogeneous. The Cray T3D is in some sense a heterogeneous parallel system which contains a vector parallel computer C90 as the front end and the massively parallel section of T3D. (The necessity of buying the front-end was evidently not a marketing plus: the

T3E does not need one.) A future parallel system may contains a cluster of machines of various computation power from workstations to tightly coupled parallel machines. The scheduling problem will inevitably be much harder on a heterogeneous system because of the different speed and memory capacity of its node processors.

More than 1000 so called supercomputers have been installed worldwide. In US, parallel machines have been installed and used at national research labs (Los Almos National Laboratory, Sandia National Labs, Oak Ridge National Laboratory, Lawrence Livermore National Laboratory, NASA Ames Research Center, US Naval Research Laboratory, DOE/Battis Atomic Power Laboratory, etc) supercomputing centers (Minnesota Supercomputer Center, Urbana-Champaign NCSA, Pittsburgh Supercomputing Center, San Diego Supercomputer Center, etc) US Government, and commercial companies (Ford Motor Company, Mobil, Amoco) and major universities. Machines from different supercomputing companies look different, are priced differently, and are named differently. Here are the names and birthplaces of some of them.

- Cray T3E (MIMD, distributed memory, 3D torus, uses Digital Alpha microprocessors), C90 (vector), Cray YMP, from Cray Research, Eagan, Minnesota.
- Thinking Machine CM-2 (SIMD, distributed memory, almost a hypercube) and CM-5 (SIMD and MIMD, distributed memory, Sparc processors with added vector units, fat tree) from Thinking Machines Corporation, Cambridge, Massachusetts.
- Intel Delta, Intel Paragon (mesh structure, distributed memory, MIMD), from Intel Corporations, Beaverton, Oregon. Based on Intel i860 RISC, but new machines based on the P6. Recently sold world's largest computer (over 6,000 P6 processors) to the US Dept of Energy for use in nuclear weapons stockpile simulations.
- IBM SP-1, SP2, (clusters, distributed memory, MIMD, based on IBM RS/6000 processor), from IBM, Kingston, New York.
- MasPar, MP-2 (SIMD, small enough to sit next to a desk), by MasPar, Santa Clara, California.
- KSR-2 (global addressable memory, hierarchical rings, SIMD and MIMD) by Kendall Square, Waltham, Massachusetts. Now out of the business.
- Fujitsu VPX200 (multi-processor pipeline), by Fujitsu, Japan.
- NEC SX-4 (multi-processor vector, shared and distributed memory), by NEC, Japan.
- Tera MTA (MPP vector, shared memory, multithreads, 3D torus), by Tera Computer Company, Seattle, Washington. A novel architecture which uses the ability to make very fast context switches between threads to hide latency of access to the memory.
- Meiko CS-2HA (shared memory, multistage switch network, local I/O device), by Meiko Concord, Massachusetts and Bristol UK.
- Cray-3 (gallium arsenide integrated circuits, multiprocessor, vector) by Cray Computer Corporation, Colorado Spring, Colorado. Now out of the business.

Chapter 9

Parallel Programming

A parallel language must provide mechanisms for implementing parallel algorithms, i.e., to specify various levels of parallelism and define parallel data structures for distributing and sharing information among processors.

Most current parallel languages add parallel constructs for standard sequential languages. Different parallel languages provide different basic constructs. The choice largely depends on the parallel computing model the language means to support.

There are at least three basic parallel computation models other than vector and pipeline model: *data parallel*, *message passing*, and *shared memory task parallel*.

9.1 Programming style

Data parallel vs. message passing. Explicit communication can be somewhat hidden if one wants to program in a data parallel style; it's something like SIMD: you specify a single action to execute on all processors. Example: if `A`, `B` and `C` are matrices, you can write `C=A+B` and let the compiler do the hard work of accessing remote memory locations, partition the matrix among the processors, etc.

By contrast, explicit message passing gives the programmer careful control over the communication, but programming requires a lot of knowledge and is much more difficult (as you probably understood from the previous section).

There are uncountable other alternatives, still academic at this point (in the sense that no commercial machine comes with them bundled). A great deal of research has been conducted on multithreading; the idea is that the programmer expresses the computation graph in some appropriate language and the machine executes the graph, and potentially independent nodes of the graph can be executed in parallel. Example in Cilk (multithreaded C, developed at MIT by Charles Leiserson and his students):

```
thread int fib(int n)
{
    if (n<2)
        return n;
    else {
        cont int x, y;
        x = spawn fib (n-2);
        y = spawn fib (n-1);
```

```
            sync;
            return x+y;
        }
    }
```

Actually Cilk is a little bit different right now, but this is the way the program will look like when you read these notes. The whole point is that the two computations of `fib(n-1)` and `fib(n-2)` can be executed in parallel. As you might have expected, there are dozens of multithreaded languages (functional, imperative, declarative) and implementation techniques; in some implementations the thread size is a single-instruction long, and special processors execute this kind of programs. Ask Arvind at LCS for details.

Writing a good HPF compiler is difficult and not every manufacturer provides one; actually for some time TMC machines were the only machines available with it. The first HPF compiler for the Intel Paragon dates December 1994.

Why SIMD is not necessarily the right model for data parallel programming. Consider the following Fortran fragment, where `x` and `y` are vectors:

```
        where (x > 0)
             y = x + 2
        elsewhere
             y = -x + 5
endwhere
```

A SIMD machine might execute both cases, and discard one of the results; it does twice the needed work (see why? there is a single flow of instructions). This is how the CM-2 operated.

On the other hand, an HPF compiler for a SIMD machine can take advantage of the fact that there will be many more elements of `x` than processors. It can execute the `where` branch until there are no positive elements of `x` that haven't been seen, then it can execute the `elsewhere` branch until all other elements of `x` are covered. It can do this provided the machine has the ability to generate independent memory addresses on every processor.

Moral: even if the programming model is that there is one processor per data element, the programmer (and the compiler writer) must be aware that it's not true.

9.2 Data Parallel Programming in High Performance Fortran

The idea in data parallel programming is to make parallel programming look almost the same as sequential programming. As in a sequential program, there is one thread of control. There can be no race conditions, and the results of a program are only indeterminate if the programmer wants them to be (via a random number generator, for example). There are no synchronization constructs. There is only one "name space" for variables; the declarations of variables in a data parallel program describe their actual shape, and they are accessible on all processors. Unlike other parallel programming dialects, like message passing, there are no private variables that are seen only by an owning processor. Users of CM Fortran and Maspar Fortran have found data parallel programming to be quite useful and relatively easy, compared to lower level parallel programming.

As you know, most scientific software is written in Fortran, and for good reasons: stable and effective compilers, a good base of libraries, reasonable support for single and double precision and complex arithmetic, and for multidimensional arrays, and good performance on uniprocessors. Programmers also hate Fortran, and for equally good reasons: no dynamic storage allocation, no function prototypes, no reasonable scoping and no real global variables, no structs, no recursion, no standard random number generator or system clock. Fortunately, Fortran 90 *has* all these desirable features. Some of the bizarre and archaic features of Fortran are labeled "obsolescent" in Fortran 90 and will be purged in Fortran 95 and later revisions. In addition, Fortran 90 allows operations on whole arrays and sections of arrays, greatly improved facilities for passing arrays or subarrays to subprograms, as well as array-valuedi constants, array-valued functions, and a number of new intrinsic functions and subroutines that operate on arrays. The array features are a simple way to specify data parallel computation as the concurrent application of a function to all the elements of an array.

High Performance Fortran (HPF) was developed during 1992 – 1993 by a group of volunteers that included representatives of users, researchers, and vendors of parallel machines. The goal was to define a standard, non-proprietary dialect of Fortran 90 for distributed memory machines that would allow programmers to write portable, high-level data parallel code. The HPF standard can be downloaded; the URL is
http://www.erc.msstate.edu/hpff/home.html. A good reference is "The High Performance Fortran Handbook," by Charles Koelbel, David Loveman, Robert Schreiber, Guy L. Steele Jr., and Mary Zosel, published by MIT Press.

HPF is Fortran 90 with a few additions. The added language features are a FORALL construct and a new attribute for subprograms, PURE. In addition, in HPF the mapping of the data to the processors can be specified.

Here is how the mapping of data is specified. All mappings are described by what are technically comments in the program. Fortran 90 comments begin with `!`. HPF directives begin with the string `!HPF$`. Thus, a Fortran 90 compiler that is liberal enough to allow `FORALL` and `PURE` (and these are to be included in the next generation, Fortran 95) will accept HPF programs! You can debug them on one processor this way. In fact, DEC has taken the obvious step of having one compiler, which accepts both Fortran 90 and HPF programs.

Thus the data mapping directives *do not change the meaning of the program.* By "meaning" I mean the relation between the output and the input. Changing the mapping of the data has no effect on *what* is computed, only on *where* it is computed.

First, one can give the machine a shape, as a multidimensional processor grid. Example:

```
!HPF$ PROCESSORS PROCS(2, NUMBER_OF_PROCESSORS() / 2)
```

`NUMBER_OF_PROCESSORS()` is an HPF intrinsic that returns, you guessed it, the number of processors that the program is running on. So we're saying with the statement above that `PROCS` is a virtual grid of processors that is $2 \times p/2$. We use `PROCS` to describe the mapping of data with the `DISTRIBUTE` directive. Example:

```
      REAL A(10,20), B(10,20,10)
!HPF$ DISTRIBUTE A(BLOCK, CYCLIC) ONTO PROCS
!HPF$ DISTRIBUTE B(BLOCK, *, CYCLIC)
```

A(i,j) is mapped to processor PROCS(1 + (i-1)/5, 1 + mod(j, p/2)). The array B is mapped to *some* two dimensional processors arrangement, whose shape is determined by the compiler. B(i,j,k) is mapped to a processor that depends on i and k, but not j — the second axis of B is mapped to processor memory. Block mappings chop an array axis up into contiguous hunks that are given one to each processor along an axis of the processor grid. Cyclic mappings deal the array out to the processors just as one deals cards to card players. Finally, HPF allows `CYCLIC(k)` mappings, where k is an integer, which deal out contiguous groups of k indices. So, if a 10 element array has a cyclic(3) distribution onto 2 processors, then processor 1 gets array elements $1, 2, 3, 7, 8$, and 9 and processor 2 gets array elements $4, 5, 6$, and 10.

One may replicate data and one may align arrays with other arrays using the ALIGN directive. Example:

```
        REAL A(10,20), C(10), D(20), E(5,5), F(20,10), S
!HPF$   ALIGN C(I) WITH A(I, 4)       ! C aligned to 4th column
!HPF$   ALIGN D(J) WITH A(*, J)       ! Replicate D across the rows of A
!HPF$   ALIGN E(I,J) WITH A(2*I, 4*J) ! Non-unit stride
!HPF$   ALIGN F(J,I) WITH A(I,J)      ! Note the transposition
!HPF$   ALIGN WITH A(*,*) :: S        ! Replicate the scalar variable.
!HPF$   DISTRIBUTE A(BLOCK, BLOCK)    ! An align target, but not an
                                      ! alignee, may be distributed.
                                      ! The aligned elements
                                      ! are "carried along".
```

Here you see a Fortran 90 variant of a declaration in which all the attributes of an object or objects are listed, then a double colon (::) then a list of the objects that have those attributes. I like it, because I can go to one place to find all that I need to know about a given variable. HPF allows this style in directives too, as the example shows.

If we wanted to get the same mapping of C, D, E, and F, but didn't have a convenient 10×20 array like A to align them to, we can use, instead, an HPF thing called a *template*, which is an array without any data, just subscripts! (Guy Steele says that if an array is a cat, then a template is a Cheshire Cat and the index set is its smile. :$\hat{)}$)) Thus:

```
!HPF$ TEMPLATE, DIMENSION(10,20) :: T
!HPF$ ALIGN E(J,I) WITH T(I,J)
!HPF$ DISTRIBUTE T(BLOCK, BLOCK)
```

Where does one put the mapping directives, and what is their scope? In general, they go in the same place one puts variable declarations. The mappings apply for the life of the object. Some programmers, however, want to be able to change a mapping during the computation, and HPF allows this. To do so, there are "executable" forms called REALIGN and REDISTRIBUTE that may appear in the executable part of a program. When control reaches the directive, the indicated remapping takes place. Any object that is remapped this way, wither directly or indirectly (by virtue of its alignment to a redistributed target) has to be given the DYNAMIC attribute. Example:

```
!HPF$   TEMPLATE, DYNAMIC, DIMENSION(2*NUMBER_OF_PROCESSORS()) :: T
!HPF$   DISTRIBUTE T(CYCLIC)
!HPF$   ALIGN (I) WITH A(I) :: B, C
!HPF$   ALIGN (I) WITH T(I) :: A
!HPF$   DYNAMIC :: A, B, C

! Executable Statements ....

!HPF$   REDISTRIBUTE T(BLOCK)  !Remaps A, B, and C
!HPF$   REALIGN A(I) WITH T(*) !Replicates A
```

Note that A, B, and C are aligned to T, although in the case of B and C, two directives are used to specify the alignment. Alignment is always simplified to an alignment to an *ultimate align target.*

This, the redistribution of T carries all three arrays with it. The realignment of A does not affect B or C. Finally, note that A, B, and C are all required to have the `DYNAMIC` attribute, even though B and C do not explicitly appear in a `REALIGN`.

9.2.1 Parallelism

There are some instances in which the Fortran 90 array syntax cannot express a simple data parallel operation, or at least not conveniently. One of the most important is when each processor uses the elements of an array that it owns as indices into some other array that it owns or of which it owns a copy. For these, one can use the HPF `FORALL` assignment statement. Example

```
      INTEGER, PARAMETER :: NPROCS = NUMBER_OF_PROCESSORS()
      REAL TABLE(NPROCS, 100), RESULT(NPROCS)
      INTEGER INDEX(NPROCS)
!HPF$ DISTRIBUTE (CYCLIC) :: INDEX, RESULT
!HPF$ DISTRIBUTE (CYCLIC,*) :: TABLE
      FORALL (P = 1:NPROCS) RESULT(P) = TABLE(P, INDEX(P))
```

Note that the Fortran 90 expression `TABLE(:,INDEX)` does not produce the rank-one result we need. Another `FORALL` example:

```
      FORALL (I = 1:N) A(I,I) = 2*I  !Access a diagonal of an array
```

The right hand sides of all instances of the `FORALL` are evaluated before any of the left hand sides is changed. The following are therefore equivalent:

```
      A(2:N) = A(1:N-1)
      FORALL (I = 1:N-1) A(I+1) = A(I)
```

9.2.2 Intrinsics

Fortran 90's intrinsic subprograms are a big improvement on Fortran 77. There are random number and system clock subroutines, array inquiry functions, and many array-valued operations, like `SUM`, `TRANSPOSE`, and `MATMULT`.

A few important data-parallel array operators were left out of Fortran 90, and are added to HPF in the HPF library. These include parallel prefix functions, combining send functions, and sorting.

Here is an example of parallel prefix. Suppose we want to enumerate the elements of a logical vector that are true. This is accomplished by `COUNT_PREFIX`. In Fortran 90, `(/ 1, 2, 3 /)` is a rank one integer array constant of size 3. Let `T` denote the value true and `F` denote false. Then `COUNT_PREFIX( (/F, T, T, F, T, F/) )` has the value `(/ 0, 1, 2, 2, 3, 3)`.

The combining send functions perform a scatter, much as the Fortran 90 vector-valued subscript does if used on the left side of an assignment statement:

```
      INTEGER, DIMENSION(:) :: A, B, V
      B(V) = A
```

The Fortran 90 fragment is meaningful only when the elements of `V` are all different, and are all legitimate subscripts for `B`. What if there are duplicate elements of `V`, so that several elements of `A` are associated with one element of `B`? If we specify that this element of `B` is to get, for example, the sum of the associated elements of `A`, then the program again makes sense. This is accompished with

```
B = SUM_SCATTER(ARRAY = A, BASE = B, INDX1 = V)
```

Note the use of Fortran 90 keyword arguments (which can occur in any order). The result of any of the scatter functions has the same shape as its BASE argument. The value of an element of the result is the corresponding element of the base unless it is associated with a set of elements of ARRAY. Element k of the result is associated with and (in the case of SUM_SCATTER is set to the sum of the elements ARRAY(i) for which V(i) is equal to k.

9.2.3 Extended data parallelism

Is data parallelism the same as array-based parallelism? Some think so, but this is a misconception. The key idea is the application of a function simultaneously to a collection of arguments. HPF allows one to specify this in a very flexible way with the INDEPENDENT DO loop

```
!HPF$ INDEPENDENT
   DO I = 1, 1024
   ...
   ENDDO
```

The independent directive does not change what is computed. It just advises the compiler that the loop iterations may be executed in parallel, with no contention for or race conditions involving shared variables. When the use of some variable by multiple iterations of a loop renders this false, we can still use the INDEPENDENT directive, as long as the use of the variable is essentially local to each iteration and it does not carry information from one iteration to another. For example,

```
DO I = 1, N
   DO J = 1, 3
   ...
   ENDDO
ENDDO
```

Ordinarily, the variable J is not used to carry information between iterations of the I loop. We can assert that the outer loop is independent *if* the variable J is made local to each outer iteration by use of the directive

```
!HPF$ INDEPENDENT, NEW J
```

just before the DO I statement; this has the same effect as replacing the shared scalar J with an array, and using J(I) at iteration I, but it uses less storage.

9.3 Message Passing

In the SISD, SIMD, and MIMD computer taxonomy, SISD machines are conventional uniprocessors, SIMD are single instruction stream machines that operate in parallel on ensembles of data, like arrays or vectors, and MIMD machines have multiple active threads of control (processes) that can operate on multiple data in parallel. How do these threads share data and how do they synchronize? For example, suppose two processes have a producer/consumer relationship. The producer generates a sequence of data items that are passed to the consumer, which processes them further. How does the producer deliver the data to the consumer?

If they share a common memory, then they agree on a location to be used for the transfer. In addition, they have a mechanism that allows the consumer to know when that location is full, *i.e.* it has a valid datum placed there for it by the producer, and a mechanism to read that location and change its state to empty. A full/empty bit is often associated with the location for this purpose. The hardware feature that is often used to do this is a "test-and-set" instruction that tests a bit in memory and simultaneously sets it to one. The producer has the obvious dual mechanisms.

Many highly parallel machines have been, and still are, just collections of independent computers on some sort of a network. Such machines can be made to have just about any data sharing and synchronization mechanism; it just depends on what software is provided by the operating system, the compilers, and the runtime libraries. One possibility, the one used by the first of these machines (The Caltech Cosmic Cube, from around 1984) is message passing. (So it's misleading to call these "message passing machines"; they are really multicomputers with message passing library software.)

From the point of view of the application, these computers can send a message to another computer and can receive such messages off the network. Thus, a process cannot touch any data other than what is in its own, private memory. The way it communicates is to send messages to and receive messages from other processes. Synchronization happens as part of the process, by virtue of the fact that both the sending and receiving process have to make a call to the system in order to move the data: the sender won't call send until its data is already in the send buffer, and the receiver calls receive when its receive buffer is empty and it needs more data to proceed.

Message passing systems have been around since the Cosmic Cube, about ten years. In that time, there has been a lot of evolution, improved efficiency, better software engineering, improved functionality. Many variants were developed by users, computer vendors, and independent software companies. Finally, in 1993, a standardization effort was attempted, and the result is the Message Passing Interface (MPI) standard. MPI is flexible and general, has good implementations on all the machines one is likely to use, and is almost certain to be around for quite some time. We'll use MPI in the course.

In print, the best MPI reference is the handbook *Using MPI*, by William Gropp, Ewing Lusk, and Anthony Skjellum, published by MIT Press ISBN 0-262-57104-8.

The standard is on the World WIde Web. The URL is
http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html. An updated version is at
ftp://ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95/mpi-report.ps

9.3.1 Sending and receiving

In order to get started, let's look at the two most important MPI functions, `MPI_Send` and `MPI_Recv`.

The call

```
MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
          int tag, MPI_Comm comm)
```

sends `count` items of data of type `datatype` starting at the location `buf`. In all message passing systems, the processes have identifiers of some kind. In MPI, the process is identified by its `rank`, an integer. The data is sent to the processes whose rank is `dest`. Possible values for `datatype` are `MPI_INT, MPI_DOUBLE, MPI_CHAR` *etc.* `tag` is an integer used by the programmer to allow the receiver to select from among several arriving messages in the `MPI_Recv`. Finally, `comm` is something called a communicator, which is essentially a subset of the processes. Ordinarily, message passing occurs within a single subset. The subset `MPI_COMM_WORLD` consists of all the processes in a single parallel job, and is predefined.

A receive call matching the send above is

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
          int tag, MPI_Comm comm, MPI_Status *status)
```

`buf` is where the data is placed once it arrives. `count`, an input argument, is the size of the buffer; the message is truncated if it is longer than the buffer. Of course, the receive has to be executed by the correct destination process, as specified in the `dest` part of the send, for it to match. `source` must be the rank of the sending process. The communicator and the tag must match. So must the datatype.

The purpose of the datatype field is to allow MPI to be used with heterogeneous hardware. A process running on a little-endian machine may communicate integers to another process on a big-endian machine; MPI converts them automatically. The same holds for different floating point formats. Type conversion, however, is not supported: an integer must be sent to an integer, a double to a double, *etc.*

Suppose the producer and consumer transact business in two word integer packets. The producer is process 0 and the consumer is process 1. Then the send would look like this:

```
int outgoing[2];
MPI_Send(outgoing, 2, MPI_INT, 1 100, MPI_COMM_WORLD)
```

and the receive like this:

```
MPI_Status stat;
int incoming[2];
MPI_Recv(incoming, 2, MPI_INT, 0 100, MPI_COMM_WORLD, &stat)
```

What if one wants a process to which several other processes can send messages, with service provided on a first-arrived, first-served basis? For this purpose, we don't want to specify the source in our receive, and we use the value `MPI_ANY_SOURCE` instead of an explicit source. The same is true if we want to ignore the tag: use `MPI_ANY_TAG`. The basic purpose of the status argument, which is an output argument, is to find out what the tag and source of such a received message are. `status.MPI_TAG` and `status.MPI_SOURCE` are components of the struct `status` of type int that contain this information after the `MPI_Recv` function returns.

This form of send and receive are "blocking", which is a technical term that has the following meaning. for the send, it means that `buf` has been read by the system and the data has been moved out as soon as the send returns. The sending process can write into it without corrupting the message that was sent. For the receive, it means that `buf` has been filled with data on return. (A call to `MPI_Recv` with no corresponding call to `MPI_Send` occurring elsewhere is a very good and often used method for hanging a message passing application.)

MPI implementations may use buffering to accomplish this. When send is called, the data are copied into a system buffer and control returns to the caller. A separate system process (perhaps using communication hardware) completes the job of sending the data to the receiver. Another implementation is to wait until a corresponding receive is posted by the destination process, then transfer the data to the receive buffer, and finally return control to the caller. MPI provides two variants of send, `MPI_Bsend` and `MPI_Ssend` that force the buffered or the rendezvous implementation. Lastly, there is a version of send that works in "ready" mode. For such a send, the corresponding receive must have been executed previously, otherwise an error occurs. On some systems, this may be faster than the blocking versions of send. All four versions of send have the same calling sequence.

NOTE: MPI allows a process to send itself data. Don't try it. On the SP-2, if the message is big enough, it doesn't work. Here's why. Consider this code:

```
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Send(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
```

The programmer is attempting to send data from process zero to all processes, including process zero; $4 \cdot \mathrm{count}$ bytes of it. If the system has enough buffer space for the outgoing messages, this succeeds, but if it doesn't, then the send blocks until the receive is executed. But since control does not return from the blocking send to process zero, the receive never does execute. If the programmer uses buffered send, then this deadlock cannot occur. An error will occur if the system runs out of buffer space, however:

```
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Bsend(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
```

9.3.2 Tags and communicators

Tags are used to keep messages straight. An example will illustrate how they are used. Suppose each process in a group has one integer and one real value, and we wish to find, on process zero, the sum of the integers and the sum of the reals. Lets write this:

```
itag = 100;
MPI_Send(&intvar, 1, MPI_INT, 0, itag, MPI_COMM_WORLD);
ftag = 101;
MPI_Send(&floatvar, 1, MPI_FLOAT, 0, ftag, MPI_COMM_WORLD);
/****  Sends are done.   Receive on process zero   ****/
if (myrank == 0) {
   intsum = 0;
   for (kount = 0; kount < nprocs; kount++) {
      MPI_Recv(&intrecv, 1, MPI_INT, MPI_ANY_SOURCE, itag, MPI_COMM_WORLD, &stat);
      intsum += intrecv;
   }
   fltsum = 0;
   for (kount = 0; kount < nprocs; kount++) {
      MPI_Recv(&fltrecv, 1, MPI_FLOAT, MPI_ANY_SOURCE, ftag, MPI_COMM_WORLD, &stat);
      fltsum += fltrecv;
   }
}
```

It looks simple, but there are a lot of subtleties here! First, note the use of MPI_ANY_SOURCE in the receives. We're happy to receive the data in the order it arrives. Second, note that we use two different tag values to distinguish between the int and the float data. Why isn't the MPI_TYPE filed enough? Because MPI does not include the type as part of the message "envelope". The envelope consists of the source, destination, tag, and communicator, and these must match in a send-receive pair. Now the two messages sent to process zero from some other process are guaranteed to arrive in the order they were sent, namely the integer message first. But that does not mean that all of the integer message precede all of the float messages! So the tag is needed to distinguish them.

This solution creates a problem. Our code, as it is now written, sends off a lot of messages with tags 100 and 101, then does the receives (at process zero). Suppose we called a library routine written by another user before we did the receives. What if that library code uses the same message tags? Chaos results. We've "polluted" the tag space. Note, by the way, that synchronizing the processes before calling the library code does not solve this problem.

MPI provides communicators as a way to prevent this problem. The communicator is a part of the message envelope. So we need to change communicators while in the library routine. To do this, we use `MPI_Comm_dup`, which makes a new communicator with the same processes in the same order as an existing communicator. For example

```
void safe_library_routine(MPI_Comm oldcomm)
{
   MPI_Comm mycomm;
   MPI_Comm_dup(oldcomm, &mycomm);
   <library code using mycomm for communication>
   MPI_Comm_free(&mycomm);
}
```

The messages sent and received inside the library code cannot interfere with those send outside.

9.3.3 Who am I?

On the SP-2 and other multicomputers, one usually writes one program which runs on all the processors. In order to differentiate its behavior, (like producer and consumer) a process usually first finds out at runtime its rank within its process group, then branches accordingly. The calls

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

sets `size` to the number of processes in the group specified by `comm` and the call

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

sets `rank` to the rank of the calling process within the group (from 0 up to $n-1$ where n is the size of the group). Usually, the first thing a program does is to call these using `MPI_COMM_WORLD` as the communicator, in order to find out the answer to the *big* questions, "Who am I?" and "How many other 'I's are there?".

Okay, I lied. That's the second thing a program does. Before it can do anything else, it has to make the call

```
MPI_Init(int *argc, char ***argv)
```

where `argc` and `argv` should be pointers to the arguments provided by UNIX to main(). While we're at it, let's not forget that one's code needs to start with

```
#include "mpi.h"
```

The *last* thing the MPI code does should be

```
MPI_Finalize()
```

No arguments.

Here's an MPI multi-process "Hello World":

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv) {
  int i, myrank, nprocs;
  double a = 0, b = 1.1, c = 0.90;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  printf("Hello world!  This is process %d out of %d\n",myrank, nprocs);
  if (myrank == 0) printf("Some processes are more equal than others.");
  MPI_Finalize();
}  /*  main  */
```

which is compiled and executed on the SP-2 at Ames by

```
babbage1% mpicc -O3 example.c
babbage1% a.out -procs 2
```

and produces (on the standard output)

```
  0:Hello world!  This is process 0 out of 2
  1:Hello world!  This is process 1 out of 2
  0:Some processes are more equal than others.
```

Another important thing to know about is the MPI wall clock timer:

```
double MPI_Wtime()
```

which returns the time in seconds from some unspecified point in the past.

9.3.4 Performance, and tolerance

Try this exercise. See how long a message of length n bytes takes between the call time the send calls send and the time the receiver returns from receive. Do an experiment and vary n. Also vary the rank of the receiver for a fixed sender. Does the model

$$\text{Elapsed\_Time}(n, r) = \alpha + \beta n$$

work? (r is the receiver, and according to this model, the cost is receiver independent.)

In such a model, the latency for a message is α seconds, and the bandwidth is $1/\beta$ bytes/second. Other models try to split α into two components. The first is the time actually spent by the sending processor and the receiving processor on behalf of a message. (Some of the per-byte cost is also attributed to the processors.) This is called the overhead. The remaining component of latency is the delay as the message actually travels through the machines interconnect network. It is ordinarily much smaller than the overhead on modern multicomputers (ones, rather than tens of microseconds).

A lot has been made about the possibility of improving performance by "tolerating" communication latency. To do so, one finds other work for the processor to do while it waits for a message to arrive. The simplest thing is for the programmer to do this explicitly. For this purpose, there are "nonblocking" versions of send and receive in MPI and other dialects.

Nonblocking send and receive work this way. A nonblock **send start** call initiates a send but returns before the data are out of the send buffer. A separate call to **send complete** then blocks the sending process, returning only when the data are out of the buffer. The same two-phase protocol is used for nonblocking receive. The **receive start** call returns right away, and the **receive complete** call returns only when the data are in the buffer.

The simplest mechanism is to match a nonblocking receive with a blocking send. To illustrate, we perform the communication operation of the previous section using nonblocking receive.

```
MPI_Request request;
MPI_IRecv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);
if (myrank == 0)
   for(dest = 0; dest < size; dest++)
      MPI_Send(sendbuf+dest*count, count, MPI_INT, dest, tag, MPI_COMM_WORLD);
MPI_Wait(&request, &stat);
```

`MPI_Wait` blocks until the nonblocking operation identified by the handle `request` completes. This code is correct regardless of the availability of buffers. The sends will either buffer or block until the corresponding **receive start** is executed, and all of these will be.

Before embarking on an effort to improve performance this way, one should first consider what the payoff will be. In general, the best that can be achieved is a two-fold improvement. Often, for large problems, it's the bandwidth (the βn term) that dominates, and latency tolerance doesn't help with this. Rearrangement of the data and computation to avoid some of the communication altogether is required to reduce the bandwidth component of communication time.

9.3.5 Who's got the floor?

We usually think of send and receive as the basic message passing mechanism. But they're not the whole story by a long shot. If we wrote codes that had genuinely different, independent, asynchronous processes that interacted in response to random events, then send and receive would be used to do all the work. Now consider computing a dot product of two identically distributed vectors. Each processor does a local dot product of its pieces, producing one scalar value per processor. Then we need to add them together and, probably, broadcast the result. Can we do this with send and receive? Sure. Do we want to? No. No because it would be a pain in the neck to write and because the MPI system implementor may be able to provide the two necessary, and generally useful collective operations (sum, and broadcast) for us in a more efficient implementation.

MPI has lots of these "collective communication" functions. (And like the sum operation, they often do computation as part of the communication.)

Here's a sum operation, on doubles. The variable `sum` on process `root` gets the sum of the variables `x` on all the processes.

```
double x, sum;
int root, count = 1;
MPI_Reduce(&x, &sum, count, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);
```

The fifth argument specifies the operation; other possibilities are `MPI_MAX`, `MPI_LAND`, `MPI_BOR`, ... which specify maximum, logical AND, and bitwise OR, for example.

The semantics of the collective communication calls are subtle to this extent: nothing happens except that a process stops when it reaches such a call, until *all* processes in the specified group reach it. Then the reduction operation occurs and the result is placed in the `sum` variable on the *root* processor. Thus, reductions provide what is called a *barrier* synchronization.

There are quite a few collective communication operations provided by MPI, all of them useful and important. We will use several in the assignment. To mention a few, MPI_Bcast broadcasts a vector from one process to the rest of its process group; MPI_Scatter sends different data from one process to each process in its a group; MPI_Gather is the inverse of a scatter: one process receives and concatenates data from all processes in its group; MPI_Allgather is like a gather followed by a broadcast: all processes receive the concatenation of data that are initially distributed among them; MPI_Reduce_scatter is like reduce followed by scatter: the result of the reduction ends up distributed among the process group. Finally, `MPI_Alltoall` implements a very general communication in which each process has a separate message to send to each member of the process group.

Often the process group in a collective communication is some subset of all the processors. In a typical situation, we may view the processes as forming a grid, let's say a 2d grid, for example. We may want to do a reduction operation within rows of the process grid. For this purpose, we can use MPI_Reduce, with a separate communicator for each row.

To make this work, each process first computes its coordinates in the process grid. MPI makes this easy, with

```
int nprocs, myproc, procdims[2], myproc_row, myproc_col;
MPI_Dims_create(nprocs, 2, procdims);
myproc_row = myrank / procdims[1];
myproc_col = myrank % procdims[1];
```

Next. one creates new communicators, one for each process row and one for each process column. The calls

```
MPI_Comm my_prow, my_pcol;
MPI_Comm_split(MPI_COMM_WORLD, myproc_row, 0, &my_prow);
MPI_Comm_split(MPI_COMM_WORLD, myproc_row, 0, &my_prow);
```

create them and

```
MPI_Comm_free(&my_prow);
MPI_Comm_free(&my_pcol);
```

free them. The reduce-in-rows call is then

```
MPI_Reduce(&x, &sum, count, MPI_DOUBLE, MPI_SUM, 0, my_prow);
```

which leaves the sum of the vectors `x` in the vector `sum` in the process whose rank in the group is zero: this will be the first process in the row. The general form is

```
MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm newcomm)
```

As in the example above, the group associated with `comm` is split into disjoint subgroups, one for every different value of `color`; the communicator for the subgroup that this process belongs to is returned in `newcomm`. The argument `key` determines the rank of this process within `newcomm`; the members are ranked according to their value of `key`, with ties broken using the rank in `comm`.

9.4 Message Passing in PVM

PVM is a popular message passing system that allows a user to treat a heterogeneous collection of networked machines as a single "parallel virtual machine". It was developed at Oak Ridge National Lab, beginning in 1989, and it continues to evolve and improve. It is freely available, easy to download and install, and it runs in user mode; any user can configure any set of machines on which he has accounts into a PVM using PVM. PVM handles all communication and synchronization, as well as format conversion for machines with different internal representations of integers, floating point values, and strings.

PVM allows the virtual machine to be configured dynamically: computers can be added to or deleted from the VM during the course of a computation. Very useful if someone switches off one of the machines! A PVM computation consists of a number of separate programs, called tasks, that use the PVM library to communicate. Tasks can be created and can drop out, also dynamically.

To obtain PVM and the documentation, one can use the web:

`http://www.epm.ornl.gov/pvm/pvm_home.html`

The book, *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing* by Al Geist Adam Beguelin Jack Dongarra Weicheng Jiang Robert Manchek, and Vaidy Sunderam, is the best reference on PVM. It too has a home page and can be downloaded:

`http://www.netlib.org/pvm3/book/pvm-book.html`

9.4.1 PVM Basics

Once one makes PVM, there are two relevant pieces of software. One is the pvm deamon (called pvmd). You start it on any machine that it has been compiled for, and it then allows you to give commands to add other machines to the current *configuration*, which is the virtual parallel machine. The user's command interface to PVM is called the PVM console, and there is also an X Windows version of it, XPVM.

Here is an example session, running at HP Labs. The machine will include two HP workstations:

```
hplpp3% pvm
new pvm shell
t40001
pvm> add hplpp4
1 successful
                    HOST     DTID
                  hplpp4    80000
pvm> conf
2 hosts, 1 data format
                    HOST     DTID      ARCH   SPEED
                  hplpp3    40000      HPPA    1000
                  hplpp4    80000      HPPA    1000
pvm>
```

The machine configuration commands can also be given by function calls from a running PVM application.

A PVM application is just a C or Fortran 77 program compiled and linked against the PVM library. Here is a sample:

```
#include <stdio.h>
#include "pvm3.h"

main()
{
        int cc, tid;
        char buf[100];

        printf("i'm t%x\n", pvm_mytid());

        cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

        if (cc == 1) {
                cc = pvm_recv(-1, -1);
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("from t%x: %s\n", tid, buf);

        } else
                printf("can't start hello_other\n");

        pvm_exit();
        exit(0);
}
```

This example illustrates some of PVM's outstanding features. First, once begun, this task first finds its own PVM task identifier (or *tid*) by calling the PVM library function `pvm_mytid`.

Next, it starts another PVM task, leaving it up to pvmd to decide which actual processor to run it on. The other task's executable file is called `hello_other`, the source for which appears below. Its tid is kept in order to use it for communication, later. In the `if` construct, we see the basics of PVM message passing. In PVM, a send is a three step process. First, one allocates a buffer using `pvm_initsend`; next, one packs data into the buffer using `pvm_pk*` routines, which specify datatype. When there are multiple machines in the configuration, this is where data format conversions occur. Last, afters its contents are all packed in, the buffer is sent with `pvm_send`, which gives the destination tid and a message tag. The receive is a similar process. The `pvm_recv` (which allows specification of source tid and message tag) returns a receive buffer identifier. This is then used to unpack the data with `pvm_upk*`.

Here is the code for `hello_other`. Notice the call to `pvm_parent` which returns the tid of the task that spawned this one.

```
#include "pvm3.h"

main()
{
        int ptid;
        char buf[100];
```

```
        ptid = pvm_parent();

        strcpy(buf, "hello, world from ");
        gethostname(buf + strlen(buf), 64);

        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid, 1);

        pvm_exit();
        exit(0);
}
```

After compiling and linking both programs, I can log into either of the two workstations and start `hello` from the UNIX prompt. Also, using the PVM console, I configure the two machines as above. The result is this:

```
hplpp4% hello
i'm t80001
from t40002: hello, world from hplpp3
hplpp4%
```

The ability to configure heterogeneous networks and use them as a parallel processor is a very impressive capability. It's what makes PVM so popular. As a message passing dialect, however, PVM lacks the power and sophistication of MPI: it has only one send mode, no asynchronous receive, no communication contexts, and limited collective communication functions. Thus, we expect MPI and PVM to evolve towards one another, each importing the strengths of the other in time.

Performance is another issue. PVM is a user application, and as such it is forced to use the communication facilities of UNIX, which add latency and limit bandwidth. Vendor supported MPI implementations appear to have significantly better performance at this time.

9.5 More on Message Passing

9.5.1 Nomenclature

- Performance:

  Latency: the time it takes to send a zero length message (overhead)
  Bandwidth: the rate at which data can be sent

- Synchrony:

  Synchronous: sending function returns when a matching receiving operation has been initiated at the destination.
  Blocking: sending function returns when the message has been safely copied.
  Non-blocking (asynchronous): sending function returns immediately. Message buffer must not be changed until it is safe to do so

- Miscellaneous:

  Interrupts: if enabled, the arrival of a message interrupts the receiving processor
  Polling: the act of checking the occurrence of an event
  Handlers: code written to execute when an interrupt occurs
  Critical sections: sections of the code which cannot be interrupted safely
  Scheduling: the process of determining which code to run at a given time
  Priority: a relative statement about the ranking of an object according to some metric

9.5.2 The Development of Message Passing

- Early Multicomputers:
  UNIVAC
  Goodyear MPP (SIMD.)
  Denelcor HEP (Shared memory)

- The Internet and Berkeley Unix (High latency, low bandwidth)

  Support for communications between computers is part of the operating system (sockets, ports,remote devices)
  Client/server applications appear

- Parallel Machines (lowest latency,high bandwidth - dedicated networks)

  Ncube
  Intel (Hypercubes, Paragon)
  Meiko (CS1,CS2)
  TMC (CM-5)

- Workstation clusters (lower latency, high bandwidth, popular networks)

  IBM (SP1,2) - optional proprietary network

- Software (libraries):

  CrOS, CUBIX, NX, Express
  Vertex
  EUI/MPL (adjusts to the machine operating system)
  CMMD
  PVM (supports heterogeneous machines; dynamic machine configuration)
  PARMACS, P4, Chamaleon
  MPI (analgum of everything above and adjusts to the operating system)

- Systems:

  Mach
  Linda (object-based system)

9.5.3 Machine Characteristics

- Nodes:

  CPU, local memory,perhaps local I/O

- Networks:

  Topology: Hypercube,Mesh,Fat-Tree, other
  Routing: circuit, packet, wormhole, virtual channel random
  Bisection bandwidth (how much data can be sent along the net)
  Reliable delivery
  Flow Control
  "State" : Space sharing, timesharing, stateless

- Network interfaces: Dumb: Fifo, control registers
  Smart: DMA (Direct Memory Access) controllers
  Very Smart: handle protocol managememt

9.5.4 Active Messages

[Not yet written]

Chapter 10

Modeling Parallel Algorithms

10.1 Modeling the Performance of Parallel Algorithms

To model and evaluate a parallel algorithm we need to understand the factors that determine performance and efficiency. Mathematical performance models are very useful in clarifying the importance of the many factors – among them communication cost, sequential bottlenecks, and load balance – that determine performance.

We begin by making some definitions. Two well-known performance measures of parallel algorithms are *speedup* and *efficiency*. Suppose a parameter n describes the size of a problem. For example, n can be the number of particles in a particle simulation problem, or n can be the number of nonzero entries of a sparse matrix. Given a parallel algorithm, implemented on some real or modeled machine, we get the solution on one processor in time $T(n,1)$ and on p processors in time $T(n,p)$. Then the speedup is

$$S(n,p) = \frac{T(n,1)}{T(n,p)}$$

and the efficiency is

$$E(n,p) = S(n,p)/p.$$

Note that both speedup and efficiency are functions of both independent variables. If we fix n, then ordinarily speedup is a monotonically increasing function of p, but is less than p, and in fact does not even grow linearly in p. Thus efficiency drops for fixed n with increasing p. On the other hand, large problems are good for parallel machines: if we fix p, efficiency and speedup usually increase with increasing n.

Why does this happen? A very simple model to explain the behavior of the speedup and efficiency functions was proposed long ago by Amdahl; it is now known as Amdahl's law. The notion is this. Suppose the time spent by one processor can be decomposed into two parts:

$$T(n,1) = T_{par}(n) + T_{seq}(n)$$

The first component is the time T_p spent doing work that is completely parallel, in the sense that p processors will speed this part up by a factor of p. The residual part is completely sequential; the work it represents goes no faster on p processors than on one. Then what is the speedup on p processors? Clearly under Amdahl's assumptions

$$T(n,p) = T_{par}(n)/p + T_{seq}(n)\,.$$

Define the sequential fraction $f = f(n) = T_seq(n)/T(n,1)$; then the speedup and efficiency are

$$S(n,p) = \frac{p}{1 + f(p-1)}$$

and

$$E(n,p) = \frac{1}{1 + f(p-1)}.$$

Figure ???? shows the efficiency function for $f \in \{.001, .01, .1\}$. As can be seen clearly, we get good efficiency up to a parallelism $p = O(1/f)$. Amdahl's law can explain the behavior we described above.

While the logic of Amdahl's argument is irrefutable, there are nevertheless two ways in which it can mislead.

First, It is sometimes argued that all applications have an inherent fraction f that is sequential, and so the resulting drop-off in speedup and efficiency for large p limits the use of parallelism. Much more common, however, is the situation in which f drops like $o(n^{-k})$ for some $k > 0$. For example, if we consider matrix product, the longest path of dependences has length $\log n$ and hende the sequential fraction is $o(n^{-k})$ for any $k < 3$. For gaussian elimination, $f(n) = O(n^{-2})$. Realistic applications are much more complicated in this respect than these kernels, but the fact remains that the sequential part of an algorithm may become vanishingly small for large problems. One may argue that in the "real world," one has to solve a given fixed problem, and so n is fixed. But then so is p! One usually scales up the machine in order to handle larger problems. So the study of scaled speedup and efficiency, in which the parameter n enters, is entirely appropriate.

The second misleading notion in the Amdahl analysis is the implicit assumption that linear speedup is necessary in order to make highly parallel machines cost-effective. The problem with this is that the cost of a machine isn't linear in p. More realistically, a machine with p processors costs $C_0 + C_1 p$, where C_0, the cost of the system memory, cabinet, software, power supplies, air conditioning, marketing and sales expenses, and so forth, is quite significant. Thus, a better model of cost effectiveness would be

$$\mathcal{E}(n,p) = \frac{(C_0 + C_1)T(n,1)}{(C_0 + C_1 p)T(n,p)}.$$

Now, it is quite possible that the cost effectiveness of a parallel machine may exceed that of its one-processor version.

10.1.1 Matrix Vector Product and Communication Cost Models

Parallel computation of the product $y = Ax$ of a matrix $A = [a_{ij}]$ and a vector x turns out to be really important. It is the "inner loop" computation in all iterative methods for solving the linear system $Ax = b$. It occurs in the inner loop of all explicit methods for time-dependent PDEs (and in explicit integrators for the linear system of ODEs $y\prime(t) = Ay(t) + b(t)$). Moreover, from the parallel implementation viewpoint, it is identical to the computation at the heart of a lot of important parallel graph algorithms. For now, we'll restrict our attention to square matrices with symmetric zero structure. Recall that the graph of A, $G = (V,E)$, has a vertex for every row, and an undirected edge from vertex i to vertex j if a_{ij} and a_{ji} are nonzero. Now, since

$$y_i = (Ax)_i =_j a_{ij}x_j =_{(i,j)\in E} a_{ij}x_j,$$

we can view the matrix vector product as a "graph" operation" as follows. Associate x_i and y_i with vertex i of G, and a_{ij} as a weight on edge (i,j). To compute y_i, communicate the value of

x from each neighboring vertex to vertex i, multiply it by the weight of the connecting edge, and add these together with $a_{ii}x_i$.

The parallelism aspects are clear. Each element of Ax can be computed independently of all others. Each of these tasks requires a row of A and an element of y, and some subset of the elements of x, those corresponding to the nonzero elements of the given row of A. If *even more* parallelism is required, one could use several processors per element of Ax, each owning a subset of the corresponding row of A, with a sum-reduction after they compute independently. Such techniques are useful on the Maspar, with its $16,384$ processors, but not on mainstream machines with a few hundred processors.

Quite surprising and complex algorithmic ideas are required to produce good efficiency for this computation on distributed memory machines, as we shall see. The issue is communication cost. Matrix-vector is a hard problem because there is not a whole lot of arithmetic to be done: only one flop per element of the matrix. So, unlike the situation with matrix multiplication, we cannot expect to do a lot of flops per datum, unless we map the work to the processors and the data to the processors so that the data needed to do the work are already on the processor doing the work, most of the time.

10.1.2 The Concept of Scalable Algorithms

The goal of parallel algorithm research is algorithms that minimize the parallel execution time $T(n,p)$. We've found it useful to consider what happens when we let p get large. In essentially all cases, if we fix the problem size n we find that the efficiency starts to drop, and goes to zero as p grows. So, to deal with the *scalability* of an algorithm, one has to allow n to grow with p. How fast? Well, it should grow fast enough so that the efficiency does not drop off to zeros. In other words, we will ask, for a given parallel algorithm, how fast does the problem have to grow as p grows in order to obtain the bound:

$$E(n(p),p) \geq E_0 > 0 \quad \mathrm{as} p \to \infty$$

(We're essentially deriving a formula for the contours of the function E in the (n,p) plane — what one wag once called the "isogoodbars".)

Well, for most algorithms, one can find such a growth rate $n(p)$ that saves the efficiency. So are these algorithms all scalable? What would it mean if n grows so fast that each processor quickly runs out of memory? It would not be too useful. For that reason, I like the definition of scalability that requires that the growth in n required to bound E away from zero be such that the memory required *per processor* be $O(1)$ (for strict scalability) or not more that $O(\log^{\alpha} p)$ (for a weaker but useful notion of scalability.)

Back to matrix-vector product. One complication is this. We have to map the vectors x and y the same way. That's because they later will participate in operations like the DAXPY, $y \leftarrow \alpha x + y$, or the dotproduct $x^T y$, which require this *alignment* of x and y. So we might as well align them right from the beginning.

Here is the solution. For dense matrices A, the right thing to do is to map blocks of the matrix to a $\sqrt{p} \times \sqrt{p}$ processor grid. For sparse matrices A with a random structure, such as the one used in the NAS parallel conjugate gradient benchmark, the same style of mapping is best. But for sparse matrices coming from PDE applications, it's best to consider the graph of the matrix, map its vertices to the processors using a locality preserving mapping, and them map either the rows or the columns of A using the given mapping.

With dense or random sparse matrices, mapping the matrix rows doesn't lead to a scalable algorithm. Let's analyze the dense $n \times n$ case. Each processor gets n/p rows. WLOG they can be contiguous. The processor that gets a row also gets the corresponding elements of x and y. To compute its part of y it needs *all* of x. Thus, the first step is to communicate x, which begins as a *distributed* and ends up as a *replicated* vector. In message passing parlance this is called a *collect* operation, in MPI it is implemented by `MPI_Allgatherv`, and if we were writing it in HPF, we could express it as a realignment from the original alignment to a replicated alignment:

```
      REAL XDIS(N), XREP(N)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) :: T(N)
!HPF$ DISTRIBUTE XDIS(BLOCK)
!HPF$ ALIGN XREP(I) WITH T(*)    ! Replicated alignment
...
      XREP = XDIS    !  Uses MPI_Allgatherv ?
```

We'll now introduce a very important technique for the analysis and understanding of a distributed memory computation. We can get very useful lower bounds on running time by consider the volume of communication. What does the collect cost? We don't care, for analysis of scalability, about constant factors. By considering the bandwidth with which a processor can send and receive data, we see that it takes $O(n)$ time, regardless of p, to receive all of x. That's because every processor has to accept $n(p-1)/p$ elements of x — the ones it doesn't own — and can only accept data at some fixed rate. Thus, we have a lower bound on communication time that is $O(n)$, independent of p. So, since there is $O(n^2)$ work to do, we can't remain efficient with more than $O(n)$ processors. In other words, as the problem grows is size, we can increase p only as the square root of the required total memory. Thus, each processor will have to store a constant number of columns or rows, and its memory requirement will grow linearly with the machine size! This is not a scalable algorithm.

On the other hand, if we do a two-dimensional mapping of the data:

```
!HPF$ DISTRIBUTE A(BLOCK, BLOCK)
```

onto a $r \times s$ virtual grid of processors, then each processor only needs n/s elements of x and produces part of n/r elements of y. The required collects of x occur within processor columns. (MPI_Allgatherv again.) Then a summation of elements of y that each processor has computed must be done within processor rows. The MPI routine MPI_Reduce_scatter does the trick. (In MPI, we use collective communication routines with a communicator for each processor column and another communicator for each processor row – see Lecture LLLL.) We choose r and s to be $O(\sqrt{p})$. Thus, communication time is $O(n/\sqrt{p})$, much less than with the by-rows mapping. In fact, we can now take p to be $O(n^2)$. If we do this, communication volume is not the limiting factor. Both arithmetic cost and communication volume give $O(1)$ lower bounds on the parallel execution time. But the critical path (for summing the elements in rows) is $O(\log n)$ long, so it dominates. (If we take $p = O(n^2/\log n)$ then we can maintain constant efficiency, with polylog memory growth — yielding a scalable, if not strictly scalable, algorithm.)

Lewis, Payne, and van de Geijn have described a very efficient implementation of the NAS CG benchmark using this idea, and paying careful attention to the mapping of x and y too. They use a `(CYCLIC, BLOCK)` mapping of A. Element i of x and of y are mapped to the (one and only) processor in the same row as row i of A, and the same column as column i of A. Then a collect of x within processor columns and a distributed sum of y in processor rows produces y in the right place. (Reference: J. G. Lewis, D. G. Payne, and R. A. van de Geijn, "Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Computers," Scalable High Performance Computing Conference 1994 .)

Matrix vector product on the SP-2, $n = 2048$

p	Mflops (2D Map)	Mflops (1D Map: $p \times 1$)	Mflops (1D Map: $1 \times p$)
1	60.	60.	60.
2	179.	180.	125.
4	305.	422.	255.
8	665.	805.	432.
16	1120.	1153.	659.
32	2082.	959.	827.

Here are some data from the SP-2. The code is in `MPI_Examples/matvec.c`
Here you see the typical situation. The unscalable algorithm "hits a wall". On the other hand, there is a region in the (n, p) plane in which the scalable algorithm is as much as 20% slower than the "good" one-dimensional algorithm. Life is not always so simple. You might consider why the (BLOCK, *) mapping is better than the (*, BLOCK) mapping of the matrix.

10.1.3 Sparse Matrices

One idea is to treat sparse matrices the same way as dense, and map the matrix blocks. If we think in terms of the graph of the sparse matrix, then we are mapping subsets of the *edges* rather than the vertices to the processors.

If this is naively done, there may be processors that get no nonzero elements. We'll have lousy load balance. To correct this, we would first permute the rows and the columns independently to equalize the load. One could use a random permutation (this idea was pushed in A. T. Ogielski and W. Aiello, Sparse matrix computations on parallel processor arrays, *SIAM J. Scient. and Stat. Comput.* 14 (1993), pp. 519–530) which balances the load pretty well with good probability. Or one can choose the permutation by some heuristic load balancing technique. We'll discuss these in Lecture XXXXX.

The sparse matrices that arise in applications of PDE's have a property that neither full nor random sparse matrices share. It's that their graphs have small, balanced separators. This means that the graphs may be partitioned into two roughly equally large subgraphs in such a way that most edges are internal to a subgraph, and only a few connect vertices in different subgraphs. We may recursively partition the subgraphs until we have as many vertex subsets as there are processors. We then assign the subgraphs to the processors, together with the elements of x and y and the rows of A that correspond to its vertices. (An alternative is to map the corresponding columns.) The communication we require is this: each processor gets the elements of x that correspond to vertices adjacent to its vertices but mapped to other processors.

Because of the small separators, this communication is tolerable. Consider a $k \times k$ grid graph. It has $n = k^2$ vertices and can be separated into two $k \times k/2$ subgraphs with an edge separator having $k = \sqrt{n}$ edges. Map each half to one of two processors. If k is big enough, then the per-processor computation time, which is $O(k^2)$ dominates the communication time, which is $O(k)$. Got 2^q processors? No, problem, just recursively bisect the graph q times, to produce a binary tree of subgraphs with the right number of "leaf" subgraphs. If we scale the machine up to more processors, we need only scale the number of grid points proportionally to the number of processors in order to maintain constant efficiency. This is what we mean by a "scalable" algorithm.

What graphs have small separators? A famous theorem of Lipton and Tarjan asserts that any *planar* graph has an $O(\sqrt{n})$ separator into balanced subgraphs. Lots of nonplanar graphs also have $O(n^e)$ separators for $e < 1$; most notably, three dimensional finite element graphs, for which e should be around 2/3. Some recent theoretical work has characterized some sets of graphs that provably have this property. The trick is to compute those small separators. That's the topic of another lecture, by Shang-hua Teng.

10.1.4 Parallel iterative Solvers

One can write a book on solving linear systems $Ax = b$ by iterative methods, and another on parallel implementation of these. Here I want to hit on a couple of important points.

First, conjugate gradient methods have become the standard tool. In a CG method, you need to be able to do matrix vector products and vector dot products fast. We've discussed the former, and the latter are simple. So what's left to talk about?

It turns out that the most important thing one can do to efficiently solve $Ax = b$ in practice is the *change the problem!* You pick a matrix B, which has to be nonsingular, and you solve $BAx = Bb$ instead. B is called the *preconditioner*. The trick is to pick B so that fewer iterations are required to get the desired accuracy, and BAx is not much more expensive to compute than Ax. (Some prefer to solve $ABy = b$ for y, and then recover $x = By$. This if "right" preconditioning, while $BAx = Bb$ is "left" preconditioning. $BACy = Bb$ followed by $x = Cy$ is also possible.)

Often, the preconditioner is not explicitly given. For example, B may be $(LU)^{-1}$ where $LU \approx A$ is an approximate triangular factorization of A; such methods, based on "incomplete" factorization in which the zero structure of L and U is predetermined and very sparse, became quite popular in the 70s and are very effective in practice. Many other possibilities exist. One important one is to use one or more iterations of some simpler iterative method, like SOR or multi-grid, as the preconditioner. For parallel machines, polynomials in A can be used, too.

When B is implicitly given, parallelism may be lost. For example. if A is tridiagonal, and B is given by and LU decomposition, we will have some trouble, even though we converge in one iteration!

Consider SOR as a preconditioner. We need to consider the parallel complexity of the SOR method. Consider the graph of A. In SOR, you visit the vertices in some order, solving for the unknown at each vertex visited while holding fixed the unknowns at the neighboring vertices. For a grid graph (a $k \times k$ mesh) we usually think about a row-by-row sweep over the graph. If we implement this in parallel, we can visit the diagonals of the graph and relax them simultaneously! This reordering of the vertices does not change the result at all. So we can perform the sweep in $O(k)$ parallel steps. On the other hand, suppose we visit first the points (i, j) for which $i + j$ is even, then the points for which $i + j$ is odd. The even points have only odd ones as neighbors, and vice versa. So we could relax all the even points in one parallel step and all the odd ones in a second parallel step. This ordering makes SOR much more parallel. It is known as "red-black" SOR (think of a checkerboard).

How about SOR on other sparse matrices? We need to color the graph of the matrix, assigning colors to vertices so that no adjacent vertices have the same color. (Remember the four-color map theorem?) Once that's done, All vertices of a given color can be relaxed in parallel, and the number of parallel steps for an SOR sweep is the number of colors we used. (Minimizing this — the minimum is the chromatic number of the graph — is too hard to do in practice. But there is a simple, fast, greedy algorithm that does pretty well.)

What about LU preconditioners. Suppose we require that $L + U$ has the same zero structure as A — i.e. we allow no fill in the incomplete factorization. Then we can parallelize the triangular solvers that implement $B + (LU)^{-1}$ with the same technique of multicoloring. If we order the gridpoints in such a way that all the points of a given color occur consecutively in the ordering, then A, L, and U develop a very nice block structure. For each color, A has a diagonal block that is itself diagonal. Since L and U inherit the structure of A, they have these diagonal diagonal blocks as well. I never thought one could use "diagonal" twice in a row in a correct sentence. Now, during the solution of, say, $Ly = f$, one can solve simultaneously for a chunk of the vector x that corresponds to one of the diagonal blocks of L in a single parallel step. The number of steps is the number of colors, again.

The idea of multicolored approximate LU is due to Schreiber and Tang. Plassman, Jones, and Freitag have used these techniques in some very large parallel computations at Argonne, and have produced a parallel solver based on them. The URL for their work is http://www.mcs.anl.gov/Projects/blocksolve/index.html

The number of parallel steps can be further reduced by a factor of two. Consider a pair of adjacent diagonal blocks of L, wlog let them be L_{11} and L_{12}. Now consider the whole block two-by-two submatrix consisting of (in Matlab notation) `[L(1,1), 0; L(2,1), L(2,2)]`. What is its inverse? What is the structure of it? Could it be economically precomputed and used in solving $Ly = g$?

Further ideas about efficient parallel solution of spare triangular systems can be found in "Highly parallel sparse triangular solution," by Alvarado, Pothen, and Schreiber, which appears in *Graph Theory and Sparse Matrix Computation*, pp. 141–157. The IMA Volumes in Mathematics and Its Applications, Volume 56, Springer-Verlag, 1993.

10.1.5 Problems

1. The sequential time $T(n, 1)$ is a measure of the total work needed to solve a problem. On p processors, the total work that can be done is by $p \cdot T(n, p)$ and hence the overhead of parallelization $H(n, P)$ is $p\dot{T}(n, p) - T(n, 1)$. Use Amdahl's assumptions to explain the overhead function.

2. Analyze the scalability of the usual substitution algorithm for solving $Lx = b$ where L is a dense, triangular matrix.

3. Give a scalable algorithm and its analysis for parallel prefix.

4. Consider a dag (directed acyclic graph) whose nodes represent unit-cost atomic computations and whose directed edges represent precedence constraints. A parallel schedule for such a dag on p processors assigns a time $t(v)$ to every vertex v such that no more than p vertices have the same time, and predecessors are scheduled before their successors. Consider only schedules that do not have idle processors while there are tasks available for execution.

   One problem with Amdahl's analysis is that is assumes that computation dags consist of straight sequential chains and of completely precedence free flat collections of independent tasks. Real dags may have neither. Generalize Amdahl's analysis for dags. Bound the efficiency in terms of the size of the dag and the length of the longest path (the critical path).

5. Use the dag model from the previous question. Assume that vertices without predecessors represent input of data from peripheral devices and vertices without successors are outputs.

   Model execution time when, in addition to p processors there are d I/O devices each capable of one input or one output operation per unit time. Can you suggest useful concepts like speedup and efficiency with a two parameter (p, d) machine model?

Chapter 11

Primitives

11.1 Parallel Prefix

An important primitive for (data) parallel computing is the *scan operation*, also called *prefix sum* which takes an associated binary operator $\oplus$ and an ordered set $[a_1, ..., a_n]$ of n elements and returns the ordered set

$$[a_1, (a_1 \oplus a_2), ..., (a_1 \oplus a_2 \oplus ... \oplus a_n)].$$

For example,

$$\texttt{plus\_scan}([1,2,3,4,5,6,7,8]) = [1,3,6,10,15,21,28,36].$$

Notice that computing the scan of an n-element array requires $n-1$ serial operations.

Suppose we have n processors, each has a element of the array. If we are interested only in the last element, namely the total sum of the arrays, then it is easy to see how to compute it efficiently in parallel: we can just break the array recursively into two halves, and add the sums of the two halves, recursively. Associated with the computation is a complete binary tree. With n processors, this takes $O(\log n)$ steps. If we have only $p < n$ processors, we can break the array into p subarray, each has roughly $\lceil n/p \rceil$ elements. At the first step, each processor adds its own elements. The problem is then reduced to one with p elements. So we can perform the $\log p$ time algorithm. The total time is clearly $O(n/p + \log p)$ and communication only occur in the second step. For architecture like hypercube and fat tree, we can embed the complete binary tree so that the communication is performed by communication link directly.

Now we discuss a parallel method of finding *all* elements $[b_1, ..., b_n] = \oplus\_scan[a_1, ..., a_n]$ in $O(\log n)$ time assume we have n processors each has an element of the array. The following is a Parallel Prefix algorithm to compute the scan of an array.

Function $\texttt{scan}([a_i])$:

1. $c_{2i} := a_{2i-1} \oplus a_{2i}$ (all i)
2. $[b_{2i}] := \texttt{scan}([c_{2i}])$ (all i)
3. $b_{2i+1} := b_{2i} \oplus a_{2i+1}$ (all i)
   Return $[b_i]$

The total number of $\oplus$ operations performed by this algorithm is (ignoring a constant term of ± 1):

$$T_n = \overbrace{\frac{n}{2}}^{\mathrm{I}} + \overbrace{T_{n/2}}^{\mathrm{II}} + \overbrace{\frac{n}{2}}^{\mathrm{III}}$$

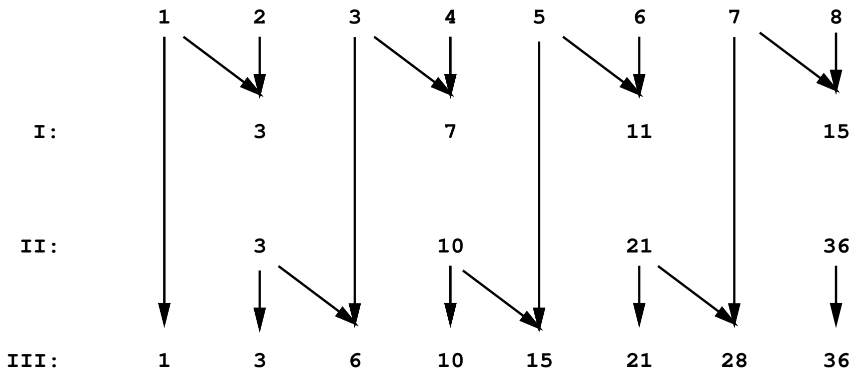


Figure 11.1: Action of the Parallel Prefix algorithm. The array in II comes from running `scan` recursively on the array in I.

$$\begin{aligned} &= n + T_{n/2} \\ &= 2n \end{aligned}$$

If there is a processor for each array element, then the number of parallel operations is:

$$\begin{aligned} T_n &= \overbrace{1}^{\mathrm{I}} + \overbrace{T_{n/2}}^{\mathrm{II}} + \overbrace{1}^{\mathrm{III}} \\ &= 2 + T_{n/2} \\ &= 2\lg n \end{aligned}$$

In practice, we usually do not have a processor for each array element. Instead, there will likely be many more array elements than processors. If we have 32 processors and an array of 32000 numbers, then each processor should store a contiguous section of 1000 array elements. Suppose we have n element and p processors. In the following assume $k = n/p$, then the procedure to compute the scan is:

1. At each processor i, compute a local scan serially, for n/p consecutive elements, giving result $[d_1^i, d_2^i, \ldots, d_k^i]$. Notice that this step vectorizes over processors.
2. Use the parallel prefix algorithm to compute
   $\mathtt{scan}([d_k^1, d_k^2, \ldots, d_k^p]) = [b_1, b_2, \ldots, b_p]$
3. At each processor $i > 1$, add b_{i-1} to all elements d_j^i.

The time taken for the will be

$$T = 2 \cdot \left(\begin{array}{c} \text{time to add and store} \\ \text{n/p numbers serially} \end{array} \right) + 2 \cdot (\log p) \cdot \left(\begin{array}{c} \text{Communication time} \\ \text{up and down a tree,} \\ \text{and a few adds} \end{array} \right)$$

11.2 Segmented Scan

We can extend the parallel scan algorithm to perform segmented scan. In *segmented scan* the original sequence is used along with an additional sequence of booleans. These booleans are used to identify the start of a new segment. Segmented scan is simply prefix scan with the additional condition the the sum starts over at the beginning of a new segment (which is indicated by a one in the string of booleans). Thus the following inputs would produce the following result when applying segmented plus scan on the array A and boolean array C.

$$\begin{aligned} A &= [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10] \\ C &= [1\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ ] \\ plus\_scan(A,C) &= [\,\underline{1\ 3\ 6\ 10}\ \ \underline{5\ 11}\ \ \underline{7}\ \ \underline{8\ 17}\ \ \underline{10}\,] \end{aligned}$$

We now show how to reduce segmented scan to simple scan. We define an operator ($\bigoplus_2$) that combines the vectors A and C (see above) which takes 2 operands x and y. These operands are simply 2 element column vectors where the first entry is from vector (A) and the second entry is from vector (C). Thus the 2-element representation of the example above is given as:

$$\begin{pmatrix}1\\1\end{pmatrix}\begin{pmatrix}2\\0\end{pmatrix}\begin{pmatrix}3\\0\end{pmatrix}\begin{pmatrix}4\\0\end{pmatrix}\begin{pmatrix}5\\1\end{pmatrix}\begin{pmatrix}6\\0\end{pmatrix}\begin{pmatrix}7\\1\end{pmatrix}\begin{pmatrix}8\\1\end{pmatrix}\begin{pmatrix}9\\0\end{pmatrix}\begin{pmatrix}10\\1\end{pmatrix}$$

The operator ($\bigoplus_2$) is defined as follows:

$$\begin{array}{c|cc} \bigoplus_2 & \begin{pmatrix}y\\0\end{pmatrix} & \begin{pmatrix}y\\1\end{pmatrix} \\ \hline \begin{pmatrix}x\\0\end{pmatrix} & \begin{pmatrix}x+y\\0\end{pmatrix} & \begin{pmatrix}y\\1\end{pmatrix} \\ \begin{pmatrix}x\\1\end{pmatrix} & \begin{pmatrix}x+y\\1\end{pmatrix} & \begin{pmatrix}y\\1\end{pmatrix} \end{array}$$

As an easy assignment, we can show that the binary operator $\bigoplus_2$ defined above is associative and satisfies has the following property: For each vector A and each boolean vector C, let AC be the 2-element representation of A and C. For each binary associative operator $\oplus$, the $\bigoplus_2$-scan(AC) gives a 2-element vector whose first row is equal to the vector computed by segmented $\oplus$-scan(A,C); whose second vector is the all 1's vector. Therefore, we can apply the parallel scan algorithm to compute the segmented scan.

Notice that the method of assigning each segment to a separate processor may results in load imbalance.

11.3 Sorting and Selection

Sorting is one of the most important operations in computations. The problem is defined as: given an array of n elements $S = \{s_1, ..., s_n\}$, transform S into an array $S' = \{s_{\pi(1)}, ..., s_{\pi(n)}\}$ by permutation such that $s_{\pi(i)} \leq s_{\pi(i+1)}$ for all $1 \leq i \leq n-1$. The rank of the element $s_{\pi(i)}$ is i. The problem of finding the rank of all element (without permuting) is called the *ranking problem.*

The *kth element selection problem* is defined as: given an array of n elements $S = \{s_1, ..., s_n\}$ and an integer k, find the element of rank k. The k *smallest elements selection problem* is to find the set of all elements whose rank is no more than k.

Sorting and selection on parallel computers have been studied intensively. Efficient algorithms are given in the forms of sorting networks, shared memory algorithms, as well as sorting algorithms on particular parallel architectures such as hypercubes, lines and meshes. We refer the reader to the book of Tom Leighton for more complete covering. Here we present an algorithm that use parallel scan for the selection problem.

The algorithm is based on "quick selection", a variant of quick sort (which is available in the Unix library). The quick selection algorithm can be described as following:

Algorithm: *Quick-Selection*(A, k)

1. Choose a random index i. We call $A[i]$ the sample.
2. Compute the number $m_<$ of the elements that are smaller than $A[i]$ and the number $m_=$ of elements that are equal to $A[i]$.
3. If $m_< < k$ and $m_< + m_= \geq k$, return $A[i]$.
4. If $m_{\leq} = m_< + m_= < k$, find the set $A_>$ of all elements that are larger than $A[i]$ and recursively call Qucik-Selection$(A_>, k - m_{\leq})$.
5. Else find the set $A_<$ of all elements that are less than $A[i]$ and recursively call Qucik-Selection$(A_<, k)$.

The expected sequential time of Quick-Selection is $O(n)$. In parallel, the key computational steps are the computation of $m_<$, $m_=$, $A_<$ or $A_>$. The $m_<$ can be found by a reduction operation as following. Generate a 0-1 array C such that $C[j] = 0$ if $A[j] < A[i]$. On a parallel machine, we need to "broadcast" $A[i]$ to all the processors. The $m_<$ is equal to the sum of elements in C which can be found by a reduction. Similarly, we can compute $m_=$. To construct $A_<$ in an array representation, we also use the C vector. By applying prefix sum on C, i.e., $B = \text{plus-scan}(C)$, if $C[j] = 1$ then $A_<[B[j]] = A[j]$. So a parallel prefix sum following by an "array indexing" will allow to construct $A_<$. Therefore, the expect number of prefix sum operations needed by Quick-Selection is $O(\log n)$.

Similar prefix sum based quick sort, shells sort, and bucket sort can be developed. Interested readers can find some of them and some other applications of prefix sum in the book by Guy Blelloch.

With the help of randomization, we can selection with constant number of parallel scans. This is the first time we publish this selection algorithm.

Select the kth smallest element.

- Select a random $m = \sqrt{n}$ elements $S = \{s_1, ..., s_m\}$.
- Sort the sample S into $s_{\pi(1)} < s_{\pi(2)} < ... < s_{\pi(m)}$ using the parallelization of the all-pairs comparison algorithm.
- Using parallel scan to find all element A' in the interval $[s_{\pi(\lfloor k/p \rfloor - 1)}, s_{\pi(\lceil k/p \rceil + 1)}]$ as well as the number $m_<$ the number of elements that are smaller than $s_{\pi(\lfloor k/p \rfloor - 1)}$ and the number $m_>$ the number of elements that are larger than $s_{\pi(\lceil k/p \rceil + 1)}$.
- From $m_<$ and $m_>$, we can decide whether the kth smallest element is the A'. If it is not in A' or $|A| > 2\sqrt{n}$, we repeat the algorithm, else we sort A' using the parallelization of the all-pairs comparison algorithm and return the proper element.

We can show that the expect number of tries in the above algorithm is bounded by a constant and hence only a constant number of prefix sums are needed.

11.4 FFT

The *Fast Fourier Transform* is perhaps the most important subroutine in scientific computing. It has applications ranging from multiplying numbers and polynomials to image and signal processing, time series analysis, and the solution of linear systems and PDEs. There are tons of books on the subject including two recent worderful ones by Charles van Loand and Briggs.

The discrete Fourier transform of a vector x is $y = F_n x$, where F_n is the $n \times n$ matrix whose entry $(F_n)_{jk} = e^{-2\pi ijk/n}$, $j, k = 0 \ldots n-1$. It is nearly always a good idea to use 0 based notation (as with the C programming language) in the context of the discrete Fourier transform. The negative exponent corresponds to Matlab's definition. Indeed in matlab we obtain `fn=fft(eye(n))`.

A good example is

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}.$$

Sometimes it is convenient to denote $(F_n)_{jk} = \omega_n^{jk}$, where $\omega_n = e^{-2\pi/n}$.

The Fourier matrix has more interesting properties than any matrix deserves to have. It is symmetric (but not Hermitian). It is Vandermonde (but not ill-conditioned). It is unitary except for a scale factor ($\frac{1}{\sqrt{n}} F_n$ is unitary). In two ways the matrix is connected to group characters: the matrix itself is the character table of the finite cyclic group, and the eigenvectors of the matrix are determined from the character table of a multiplicative group.

The trivial way to do the Fourier transform is to compute the matrix-vector multiply requiring n^2 multipilications and roughly the same number of additions. Cooley and Tukey gave the first $O(n \log n)$ time algorithm (actually the algorithm may be found in Gauss' work) known today as the FFT algorithm. We shall assume that $n = 2^p$.

The Fourier matrix has the simple property that if Π_n is an unshuffle operation, then

$$F_n \Pi_n^T = \begin{pmatrix} F_{n/2} & D_n F_{n/2} \\ F_{n/2} & -D_n F_{n/2} \end{pmatrix}, \tag{11.1}$$

where D_n is the diagonal matrix $\mathrm{diag}(1, \omega_n, \ldots, \omega_n^{n/2-1})$.

One DFT algorithm is then simply: 1) unshuffle the vector 2) recursively apply the FFT algorithm to the top half and the bottom half, then combine elements in the top part with corresponding elements in the bottom part ("the butterfly") as prescribed by the matrix $\begin{pmatrix} I & D_n \\ I & -D_n \end{pmatrix}$.

Everybody has their favorite way to visualize the FFT algorithm. For us, the right way is to think of the data as living on a hypercube. The algorithm is then, permute the cube, perform the FFT on a pair of opposite faces, and then perform the butterfly, along edges across the dimension connecting the opposite faces.

We now repeat the three steps of the recursive algorithm in index notation:

- Step 1: $i_{d-1} \ldots i_1 i_0 \to i_0 i_{d-1} \ldots i_1$
- Step 2: $i_0 i_{d-1} \ldots i_1 \to i_0 \mathrm{fft}(i_{d-1} \ldots i_1)$
- Step 3: $\to \bar{i_0} \mathrm{fft}(i_{d-1} \ldots i_1)$

$$y_j = (F_n x)_j =_{k=0}^{n-1} \omega_n^{jk} x_k$$

can be cut into the even and the odd parts:

$$y_j = {}_{k=0}^{m-1} \; \omega_n^{2jk} x_{2k} + \omega_n^j \left({}_{k=0}^{m-1} \omega_n^{2jk} x_{2k+1} \right) \;\; ;$$

since $\omega_n^2 = \omega_m$, the two sums are just $\mathrm{FFT}(x_{\mathrm{even}})$ and $\mathrm{FFT}(x_{\mathrm{odd}})$. With this remark (see Fig. 1),

$$\begin{array}{rcrcl} y_j & = & {}_{k=0}^{m-1}\omega_m^{jk} x_{2k} & + & \omega_n^j \left({}_{k=0}^{m-1} \omega_m^{jk} x_{2k+1} \right) \\ y_{j+m} & = & {}_{k=0}^{m-1}\omega_m^{jk} x_{2k} & - & \omega_n^j \left({}_{k=0}^{m-1} \omega_m^{jk} x_{2k+1} \right) \quad . \end{array}$$

Then the algorithm keeps recurring; the entire "communication" needed for an FFT on a vector of length 8 can be seen in Fig. 2

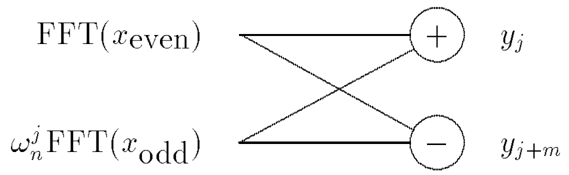


Figure 11.2: Recursive block of the FFT.

The number of operations for an FFT on a vector of length n equals to twice the number for an FFT on length $n/2$ plus $n/2$ on the top level. As the solution of this recurrence, we get that the total number of operations is $\frac{1}{2} n \log n$.

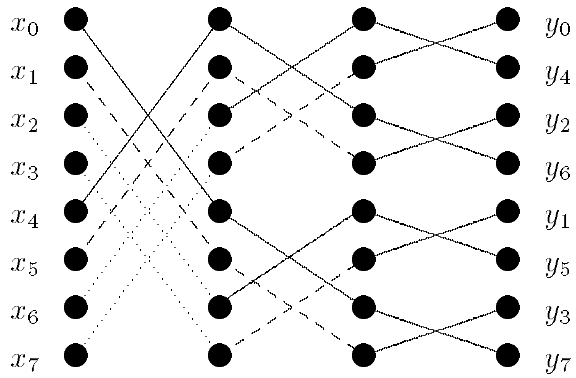


Figure 11.3: FFT network for 8 elements. (Such a network is not built in practice)

Now we analyze the data motion required to perform the FFT. First we assume that to each processor one element of the vector x is assigned. Later we discuss the "real-life" case when the number of processors is less than n and hence each processor has some subset of elements. We also discuss how FFT is implemented on the CM-2 and the CM-5.

The FFT always goes from high order bit to low order bit, i.e., there is a fundamental asymmetry that is not evident in the figures below. This seems to be related to the fact that one can obtain a subgroup of the cyclic group by alternating elements, but not by taking, say, the first half of the elements.

11.4.1 Data motion

Let $i_p i_{p-1} \dots i_2 i_1 i_0$ be a bit sequence. Let us call $i_0 i_1 i_2 \dots i_{p-1} i_p$ the **bit reversal** of this sequence. The important property of the FFT network is that if the i-th input is assigned to the i-th processor for $i \le n$, then the i-th output is found at the processor with address *the bit-reverse of* i. Consequently, if the input is assigned to processors with bit-reversed order, then the output is in standard order. The inverse FFT reverses bits in the same way.

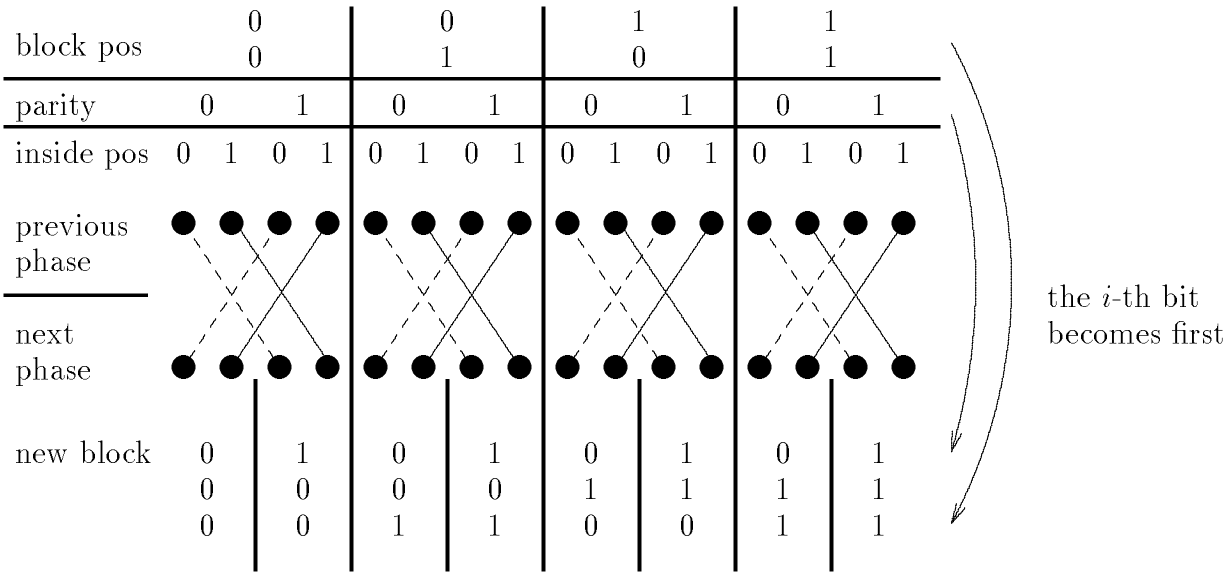


Figure 11.4: The output of FFT is in bit-reversed order.

To see why FFT reverses the bit order, let us have a look at the i-th segment of the FFT network (Fig. 3). The input is divided into parts and the current input (top side) consists of FFT's of these parts. One "block" of the input consists of the same fixed output element of all the parts. The $i-1$ most significant bits of the input address determine this output element, while the least significant bits the the part of the original input whose transforms are at this level.

The next step of the FFT computes the Fourier transform of twice larger parts; these consist of an "even" and an "odd" original part. Parity is determined by the i-th most significant bit.

Now let us have a look at one unit of the network in Fig. 1; the two inputs correspond to the same even and odd parts, while the two outputs are the possible "farthest" vector elements, they differ in the most significant bit. What happens is that the i-th bit jumps first and becomes most significant (see Fig. 3).

Now let us follow the data motion in the entire FFT network. Let us assume that the i-th input element is assigned to processor i. Then after the second step a processor with binary address $i_p i_{p-1} i_{p-2} \dots i_1 i_0$ has the $i_{p-1} i_p i_{p-2} \dots i_1 i_0$-th data, the second bit jumps first. Then the third, fourth, ..., p-th bits all jump first and finally that processor has the $i_0 i_1 i_2 \dots i_{p-1} i_p$-th output element.

11.4.2 FFT on parallel machines

In a realistic case, on a parallel machine some bits in the input address are local to one processor. The communication network can be seen in Fig. 4, left. FFT requires a large amount of commu-

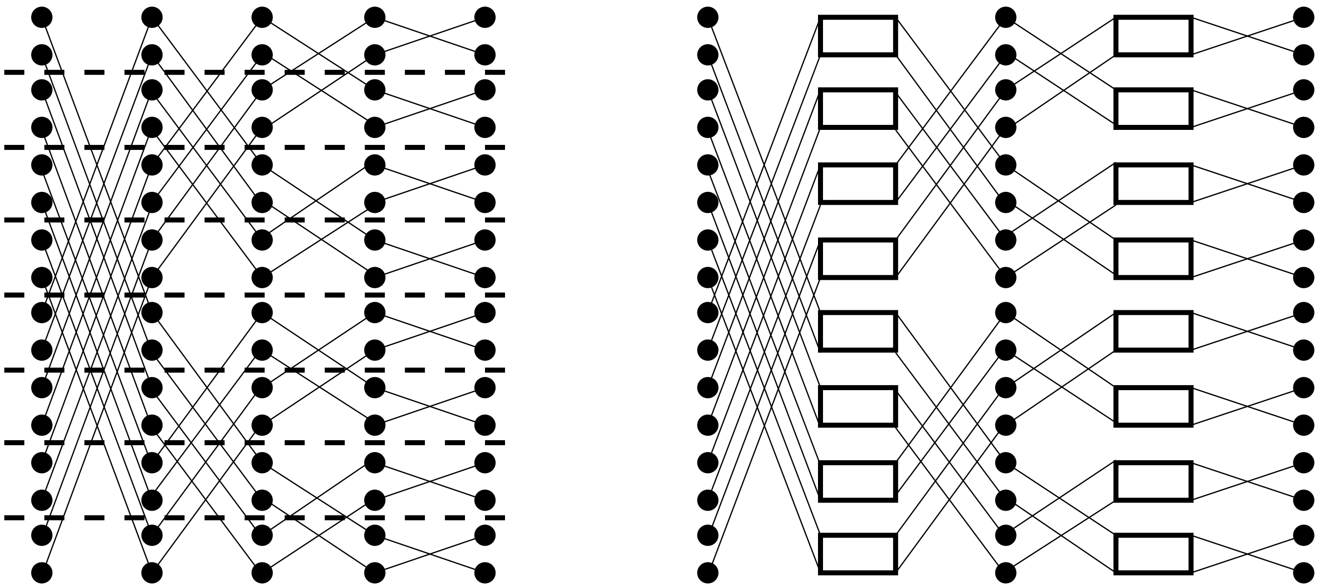


Figure 11.5: Left: more than one element per processor. Right: one box is a 4×4 matrix multiply.

nication; indeed it has fewer operations per communication than usual dense linear algebra. One way to increase this ratio is to *combine some layers* into one, as in Fig. 4, right. If s consecutive layers are combined, in one step a $2^s \times 2^s$ matrix multiplication must be performed. Since matrix multiplication vectorizes and usually there are optimized routines to do it, such a step is more efficient than communicating all small parts. Such a modified algorithm is called the *High Radix* FFT.

The FFT algorithm on the CM-2 is basically a High Radix FFT. However, on the CM-5 data motion is organized in a different way. The idea is the following: if s bits of the address are local to one processor, the last s phases of the FFT do not require communication. Let 3 bits be local to one processor, say. On the CM-5 the following data rearrangement is made: the data from the

$$i_p i_{p-1} i_{p-2} \ldots i_3 | i_2 i_1 i_0 \text{ -th}$$

processor is moved to the

$$i_2 i_1 i_0 i_{p-3} i_{p-4} \ldots i_3 | i_p i_{p-1} i_{p-2} \text{ -th!}$$

This data motion can be arranged in a clever way; after that the next 3 steps are local to processors. Hence the idea is to perform all communication at once *before* the actual operations are made.

11.4.3 Exercises

1. Verify equation (**??**).

2. Just for fun, find out about the FFT through Matlab.

   We are big fans of the `phone` command for those students who do not already have a good physical feeling for taking Fourier transforms. This command shows (and plays if you have a speaker!) the signal generated when pressing a touch tone telephone in the United States and many other countries. In the old days, when a pushbutton phone was broken apart, you could see that pressing a key depressed one lever for an entire row and another lever for an entire column. (For example, pressing 4 would depress the lever corresponding to the second row and the lever corresponding to the first column.)

To look at the FFT matrix, in a way, `plot(fft(eye(7)));axis('square')`.

3. In Matlab use the `flops` function to obtain a flops count for FFT's for different power of 2 size FFT's. Make you input complex. Guess a flop count of the form $a+bn+c\log n+dn\log n$. Remembering that Matlab's \ operator solves least squares problems, find a, b, c and d. Guess whether Matlab is counting flops or using a formula.

11.5 Matrix Multiplication

Everyone thinks that to multiply two 2-by-2 matrices requires 8 multiplications. However, Strassen gave a method, which requires only 7 multiplications! Actually, compared to the 8 multiplies and 4 adds of the traditional way, Strassen's method requires only 7 multiplies but 18 adds. Nowadays when multiplication of numbers is as fast as addition, this does not seem so important. However when we think of block matrices, matrix multiplication is very slow compared to addition. Strassen's method will give an $O(n^{2.8074})$ algorithm for matrix multiplication, in a recursive way very similar to the FFT.

First we describe Strassen's method for two block matrices:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} P_1+P_4-P_5+P_7 & P_3+P_5 \\ P_2+P_4 & P_1+P_3-P_2+P_6 \end{pmatrix}$$

where

$$\begin{aligned} P_1 &= (A_{1,1}+A_{1,2})(B_{1,1}+B_{2,2}) \ , \\ P_2 &= (A_{2,1}+A_{2,2})B_{1,1} \ , \\ P_3 &= A_{1,1}(B_{1,2}-B_{2,2}) \ , \\ P_4 &= A_{2,2}(B_{2,1}-B_{1,1}) \ , \\ P_5 &= (A_{1,1}+A_{1,2})B_{2,2} \ , \\ P_6 &= (A_{2,1}-A_{1,1})(B_{1,1}+B_{1,2}) \ , \\ P_7 &= (A_{1,2}-A_{2,2})(B_{2,1}+B_{2,2}) \ . \end{aligned}$$

If, as in the FFT algorithm, we assume that $n=2^p$, the matrix multiply of two n-by-n matrices calls 7 multiplications of $(n/2)$-by-$(n/2)$ matrices. Hence the time required for this algorithm is $O(n^{\log_2 7}) = O(n^{2.8074})$. Note that Strassen's idea can further be improved (of course, with the loss that several additions have to be made and the constant is impractically large) the current such record is an $O(n^{2.376})$-time algorithm.

A final note is that, again as in the FFT implementations, we do not recur and use Strassen's method with 2-by-2 matrices. For some sufficient p, we stop when we get $2^p \times 2^p$ matrices and use direct matrix multiply which vectorizes well on the machine.

11.6 Basic Data Communication Operations

We conclude this section by list the set of basic data communication operations that are commonly used in a parallel program.

- **Single Source Broadcast**:
- **All-to-All Broadcast**:
- **All-to-All Personalized Communication**:
- **Array Indexing or Permutation**: There are two types of array indexing: the *left* array indexing and the *right* array indexing.
- **Polyshift**: SHIFT and EOSHIFT.
- **Sparse Gather and Scatter**:
- **Reduction and Scan**:

III Multipole Methods

Differential-equation based numerical methods are very important and have been applied to a large class of scientific problems. "However, not everyone has a complete faith in differential-equation based methods. Particles methods provide them with an alternative numerical approach to formulate and solve their problems from first principles" – paraphrased from Ahmed Sameh.

Chapter 12

Particle Methods

12.1 Reduce and Broadcast: A function viewpoint

[This section is being rewritten with what we hope will be the world's clearest explnation of the fast multipole algorithm. Readers are welcome to take a quick look at this section, or pass to the next section which leads up to the multipole algorithm through the particle method viewpoint]

Imagine we have P processors, and P functions $f_1(z), f_2(z), \ldots, f_P(z)$, one per processor. Our goal is for every processor to know the sum of the functions $f(z) = f_1(z) + \ldots + f_P(z)$. Really this is no different from the reduce and broadcast situation given in the introduction.

As a practical question, how can functions be represented on the computer? Probably we should think of Taylor series or multipole expansion. If all the Taylor series or multipole expansions are centered at the same point, then the function reduction is easy. Simply reduce the corresponding coefficients. If the pairwise sum consists of functions represented using different centers, then a common center must be found and the functions must be transformed to that center before a common sum may be found.

> **Example: Reducing Polynomials** Imagine that processor i contains the polynomial $f_i(z) = (z-i)^3$. The coefficients may be expanded out as $f_i(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3$. Each processor i contains a vector (a_0, a_1, a_2, a_3). The sum of the vectors may be obtained by a usual reduce algorithm on vectors.
>
> An alternative that may seem like too much trouble at first is that every time we make a pairwise sum we shift to a common midpoint. For example,
>
> An example will go here

There is another complication that occurs when we form pairwise sums of functions. If the expansions are multipole or Taylor expansions, we may shift to a new center that is outside the region of convergence. The coefficients may then be meaningless. Numerically, even if we shift towards the boundary of a region of convergence, we may well lose accuracy, especially since most computations choose to fix the number of terms in the expansion to keep.

Difficulties with shifting multipole or Taylor Expansions

The fast multipole algorithm accounts for these difficulties in a fairly simple manner. Instead of computing the sum of the functions all the way up the tree and then broadcasting back, it saves the intermediate partial sums summing them in only when appropriate. The figure below indicates when this is appropriate.

12.2 Particle Methods: An Application

Imagine we want to model the basic mechanics of our solar system. We would probably start with the sun, somehow representing its mass, velocity, and position. We might then add each of the nine planets in turn, recording their own masses, velocities, and positions at a point in time. Let's say we add in a couple of hundred of the larger asteroids, and a few of our favorite comets. Now we set the system in motion. Perhaps we would like to know where Pluto will be in a hundred years, or whether a comet will hit us soon. To solve Newton's equations directly with more than even two bodies is intractably difficult. Instead we decide to model the system using discrete time intervals, and computing at each time interval the force that each body exerts on each other, and changing the velocities of the bodies accordingly. This is an example of an N-body problem. To solve the problem in a simple way requires $O(n^2)$ time for each time step. With some considerable effort, we can reduce this to $O(n)$ (using the fast multipole algorithm to be described below). A relatively simple algorithm the Barnes-Hut Algorithm, to be described below) can compute movement in $O(n \log(n))$ time.

12.3 Outline

- Formulation and applications
- "The easiest part": the Euler method to move bodies.
- Direct methods for force computation.
- Hierarchical methods *(Barnes-Hut, Appel, Greengard and Rohklin)*

12.4 What is N-Body Simulation?

We take n bodies (or particles) with state describing the initial position $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n \in \Re^k$ and initial velocities $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n \in \Re^k$.

We want to simulate the evolution of such a system, i.e., to compute the trajectories of each body, under an interactive force: the force exerted on each body by the whole system at a given point. For different applications we will have different interaction forces, such as gravitational or Coulombic forces. We could even use these methods to model spring systems, although the advanced methods, which assume forces decreasing with distance, do not work under these conditions.

12.5 Examples

- **Astrophysics**: The bodies are stars or galaxies, depending on the scale of the simulation. The interactive force is gravity.
- **Plasma Physics**: The basic particles are ions, electrons, etc; the force is Coulombic.

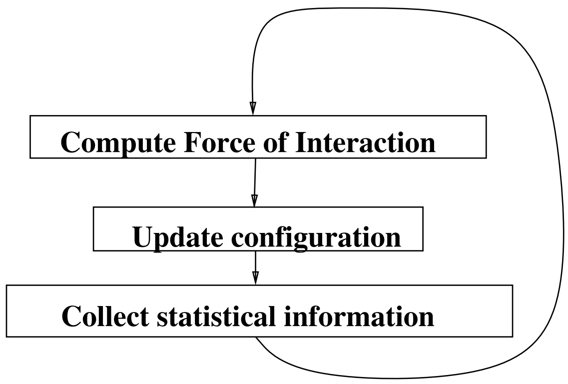


Figure 12.1: Basic Algorithm of N-body Simulation

- **Molecular Dynamics**: Particles are atoms or clusters of atoms; the force is electrostatic.
- **Fluid Dynamics**: Vortex method where particle are fluid elements (fluid blobs).

Typically, we call this class of simulation methods, the **particle methods**. In such simulations, it is important that we choose both spatial and temporal scales carefully, in order to minimize running time and maximize accuracy. If we choose a time scale too large, we can lose accuracy in the simulation, and if we choose one too small, the simulations will take too long to run. A simulation of the planets of the solar system will need a much larger timescale than a model of charged ions. Similarly, spatial scale should be chosen to minimize running time and maximize accuracy. For example, in applications in fluid dynamics, molecular level simulations are simply too slow to get useful results in a reasonable period of time. Therefore, researchers use the vortex method where bodies represent large aggregates of smaller particles. Hockney and Eastwood's book **Computer Simulations Using Particles,**, McGraw Hill (1981), explores the applications of particle methods applications, although it is somewhat out of date.

12.6 The Basic Algorithm

Figure 12.1 illustrates the key steps in n-body simulation. The step of collecting statistical information is application dependent, and some of the information gathered at this step may be used during the next time interval.

We will use gravitational forces as an example to present N-body simulation algorithms. Assume there are n bodies with masses $m_1, m_2, \ldots, m_n$, respectively, initially located at $\vec{x_1}, ..., \vec{x_n} \in \Re^3$ with velocity $\vec{v_1}, ..., \vec{v_n}$. The gravitational force exert on the ith body by the jth body is given by

$$\vec{F_{ij}} = G\frac{m_i m_j}{r^2} = G\frac{m_i m_j}{|\vec{x_j} - \vec{x_i}|^3}(\vec{x_j} - \vec{x_i}),$$

where G is the gravitational constant. Thus the total force on the ith body is the vector sum of all these forces and is give by,

$$\vec{F_i} =_{j \neq i} \vec{F_{ij}}.$$

Let $\vec{a_i} = d\vec{v_i}/dt$ be the acceleration of the body i, where where $\vec{v_i} = d\vec{x_i}/dt$. By Newton's second law of motion, we have $\vec{F_i} = m_i\vec{a_i} = m_i d\vec{v_i}/dt$.

12.6.1 Finite Difference and the Euler Method

In general, the force calculation is the most expensive step for N-body simulations. We will present several algorithms for this later on, but first assume we have already calculated the force $\vec{F_i}$ acting one each body. We can use a numerical method (such as the Euler method) to update the configuration.

To simulate the evolution of an N-body system, we decompose the time interval into discretized time steps: $t_0, t_1, t_2, t_3, ...$ For uniform discretizations, we choose a Δt and let $t_0 = 0$ and $t_k = k\Delta t$. The Euler method approximates the derivative by finite difference.

$$\begin{aligned} \vec{a_i}(t_k) &= \vec{F_i}/m_i = \frac{\vec{v_i}(t_k) - \vec{v_i}(t_k - \Delta t)}{\Delta t} \\ \vec{v_i}(t_k) &= \frac{\vec{x_i}(t_k + \Delta t) - \vec{x_i}(t_k)}{\Delta t}, \end{aligned}$$

where $1 \leq i \leq n$. Therefore,

$$\vec{v_i}(t_k) = \vec{v_i}(t_{k-1}) + \Delta t(\vec{F_i}/m_i) \tag{12.1}$$
$$\vec{x_i}(t_{k+1}) = \vec{x_i}(t_k) + \Delta t\vec{v_i}(t_k). \tag{12.2}$$

From the given initial configuration, we can derive the next time step configuration using the formulae by first finding the force, from which we can derive velocity, and then position, and then force at the next time step.

$$\vec{F_i} \to v_i(t_k) \to x_i(t_k + \Delta t) \to \vec{F_{i+1}}.$$

High order numerical methods can be used here to improve the simulation. In fact, the Euler method that uses uniform time step discretization performs poorly during the simulation when two bodies are very close. We may need to use non-uniform discretization or a sophisticated time scale that may vary for different regions of the N-body system.

In one region of our simulation, for instance, there might be an area where there are few bodies, and each is moving slowly. The positions and velocities of these bodies, then, do not need to be sampled as frequently as in other, higher activity areas, and can be determined by extrapolation. See figure 12.2 for illustration.[1]

How many floating point operations (flops) does each step of the Euler method take? The velocity update (step 1) takes $2n$ floating point multiplications and one addition and the position updating (step 2) takes 1 multiplication and one addition. Thus, each Euler step takes $5n$ floating point operations. In Big-O notation, this is an $O(n)$ time calculation with a constant factor 5.

Notice also, each Euler step can be parallelized without communication overhead. In data parallel style, we can express steps (1) and (2), respectively, as

$$\begin{aligned} V &= V + \Delta t(F/M) \\ X &= X + \Delta t V, \end{aligned}$$

where V is the velocity array; X is the position array; F is the force array; and M is the mass array. V, X, F, M are $3 \times n$ arrays with each column corresponding to a particle. The operator / is the elementwise division.

[1] In figure 12.2 we see an example where we have some close clusters of bodies, and several relatively disconnected bodies. For the purposes of the simulation, we can ignore the movement of relatively isolated bodies for short periods of time and calculate more frames of the proximous bodies. This saves computation time and grants the simulation more accuracy where it is most needed. In many ways these sampling techniques are a temporal analogue of the later discussed Barnes and Hut and Multipole methods.

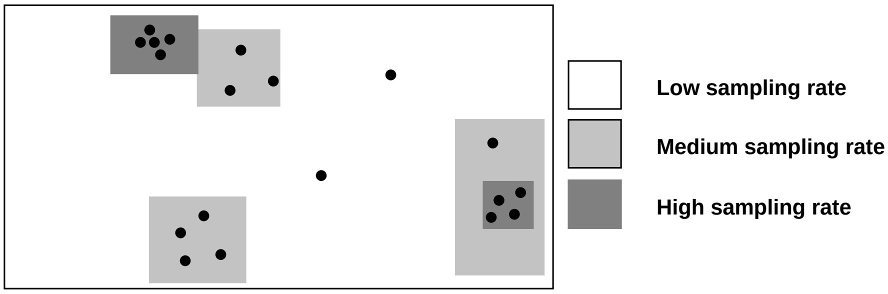


Figure 12.2: Adaptive Sampling Based on Proximity

12.7 Methods for Force Calculation

Computationally, the force calculation is the most time expensive step for N-body simulation. We now discuss some methods for computing forces.

12.7.1 Direct force calculation

The simplest way is to calculate the force directly from the definition.

$$\vec{F_{ij}} = G\frac{m_i m_j}{r^2} = G\frac{m_i m_j}{|\vec{x_j} - \vec{x_i}|^3}(\vec{x_j} - \vec{x_i}),$$

Note that the step for computing $\vec{F_{ij}}$ takes 9 flops. It takes n flops to add $\vec{F_{ij}}$ $(1 \leq j \leq n)$. Since $\vec{F_{ij}} = -\vec{F_{ji}}$, the total number of flops needed is roughly $5n^2$. In Big-O notation, this is an $O(n^2)$ time computation. For large scale simulation (e.g., $n = 100$ million), the direct method is impractical with today's level of computing power.

It is clear, then, that we need more efficient algorithms. The one fact that we have to take advantage of is that in a large system, the effects of individual distant particles on each other may be insignificant and we may be able to disregard them without significant loss of accuracy. Instead we will cluster these particles, and deal with them as though they were one mass. Thus, in order to gain efficiency, we will approximate in space as we did in time by discretizing.

12.7.2 Potential based calculation

For N-body simulations, sometimes it is easier to work with the (gravitational) potential rather than with the force directly. The force can then be calculated as the gradient of the potential.

In three dimensions, the gravitational potential at position $\vec{x}$ defined by n bodies with masses $m_1, ..., m_n$ at position $\vec{x_1}, ..., \vec{x_n}$, respectively is equal to

$$\Phi(\vec{x}) =_{i=1}^{n} G\frac{m_i}{||\vec{x} - \vec{x_i}||}.$$

The force acting on a body with unit mass at position $\vec{x}$ is given by the *gradient* of Φ, i.e.,

$$F = -\nabla\Phi(x).$$

The potential function is a sum of local potential functions

$$\phi(\vec{x}) =_{i=1}^{n} \phi_{\vec{x}_i}(\vec{x}) \qquad (12.3)$$

where the local potential functions are given by

$$\phi_{\vec{x}_i}(\vec{x}) = \frac{G * m_i}{||\vec{x} - \vec{x}_i||} \qquad \text{in } \Re^3 \tag{12.4}$$

12.7.3 Poisson Methods

The earlier method from 70s is to use Poisson solver. We work with the gravitational potential field rather than the force field. The observation is that the potential field can be expressed as the solution of a Poisson equation and the force field is the gradient of the potential field.

The gravitational potential at position $\vec{x}$ defined by n bodies with masses $m_1, ..., m_n$ at position $\vec{x_1}, ..., \vec{x_n}$, respectively is equal to

$$\Phi(\vec{x}) =_{i=1}^{n} G\frac{m_i}{|\vec{x} - \vec{x_i}|}.$$

The force acting on a body with unit mass at position $\vec{x}$ is given by the gradient of Φ:

$$\vec{F} = -\nabla\Phi(\vec{x}).$$

So, from Φ we can calculate the force field (by numerical approximation).

The potential field Φ satisfies a Poisson equation:

$$\nabla^2\Phi = \frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2} + \frac{\partial^2\Phi}{\partial z^2} = \rho(x, y, z),$$

where ρ measures the mass distribution can be determined by the configuration of the N-body system. (The function Φ is harmonic away from the bodies and near the bodies, $div\nabla\Phi = \nabla^2\Phi$ is determined by the mass distribution function. So $\rho = 0$ away from bodies).

We can use finite difference methods to solve this type of partial differential equations. In three dimensions, we discretize the domain by a structured grid.

We approximate the Laplace operator ∇^2 by finite difference and obtain from $\nabla^2\Phi = \rho(x, y, z)$ a system of linear equations. Let h denote the grid spacing. We have

$$\begin{aligned}\Phi(x_i, y_j, z_k) &= \frac{1}{h^2}(\Phi(x_i + h, y_j, z_k) + \Phi(x_i - h, y_j, z_k) + \Phi(x_i, y_j + h, z_k) \\ &\quad +\Phi(x_i, y_j - h, z_k) + \Phi(x_i, y_j, z_k + h) + \Phi(x_i, y_j, z_k - h) - 6\phi(x_i, y_j, z_k)) \\ &= \rho(x_i, y_j, z_k).\end{aligned}$$

The resulting linear system is of size equal to the number of grid points chosen. This can be solved using methods such as FFT (fast Fourier transform), SOR (successive overrelaxation), multigrid methods or conjugate gradient. If n bodies give a relatively uniform distribution, then we can use a grid which has about n grid points. The solution can be fairly efficient, especially on parallel machines. For highly non-uniform set of bodies, hybrid methods such as finding the potential induced by bodies within near distance by direct method, and approximate the potential field induced by distant bodies by the solution of a much smaller Poisson equation discretization. More details of these methods can be found in Hockney and Eastwood's book.

12.7.4 Hierarchical methods

We now discuss several methods which use a hierarchical structure to decompose bodies into clusters. Then the force field is approximated by computing the interaction between bodies and clusters

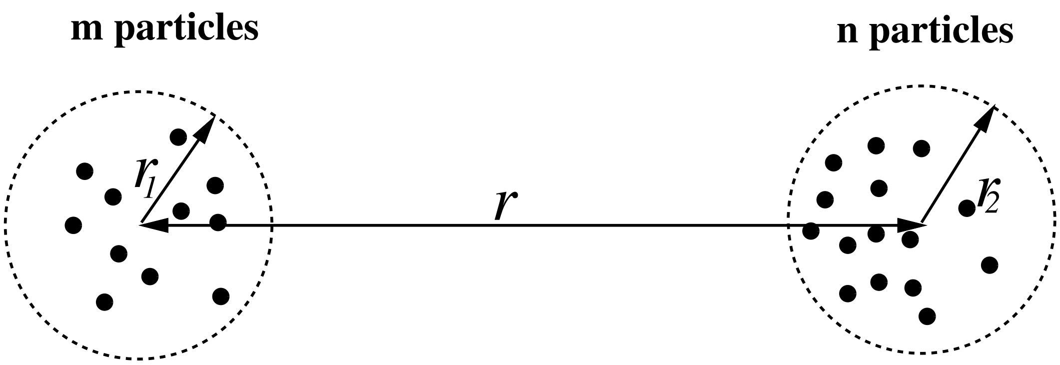


Figure 12.3: Well-separated Clusters

and/or between clusters and clusters. We will refer this class of methods *hierarchical methods or tree-code methods.*

The crux of hierarchical N-body methods is to decompose the potential at a point x, $\phi(x)$, into the sum of two potentials: $\phi_N(x)$, the potential induced by "neighboring" or "near-field" particles; and $\phi_F(x)$, the potential due to "far-field" particles [5, 41]. In hierarchical methods, $\phi_N(x)$ is computed exactly, while $\phi_F(x)$ is computed approximately.

The approximation is based on a notion of well-separated clusters [5, 41]. Suppose we have two clusters A and B, one of m particles and one of n particles, the centers of which are separated by a distance r. See Figure 12.3.

Suppose we want to find the force acting on all bodies in A by those in B and vice versa. A direct force calculation requires $O(mn)$ operations, because for each body in A we need to compute the force induced by every body in B.

Notice that if r is much larger than both r_1 and r_2, then we can simplify the calculation tremendously by replacing B by a larger body at its center of mass and replacing A by a larger body at its center of mass. Let M_A and M_B be the total mass of A and B, respectively. The center of mass c_A and c_B is given by

$$\begin{aligned} c_A &= \frac{{}_{i\in A} m_i x_i}{M_A} \\ c_B &= \frac{{}_{j\in B} m_j x_j}{M_B}. \end{aligned}$$

We can approximate the force induced by bodies in B on a body of mass m_x located at position s by viewing B as a single mass M_B at location c_B. That is,

$$F(x) \approx \frac{G m_x M_B (x - c_B)}{||x - c_B||^3}.$$

Such approximation is second order: The relative error introduced by using center of mass is bounded by $(\max(r_1, r_2)/r)^2$. In other words, if $f(x)$ be the true force vector acting on a body at x, then

$$F(x) = f(x)\left(1 + O\left(\left(\frac{\max(r_1, r_2)}{r}\right)^2\right)\right).$$

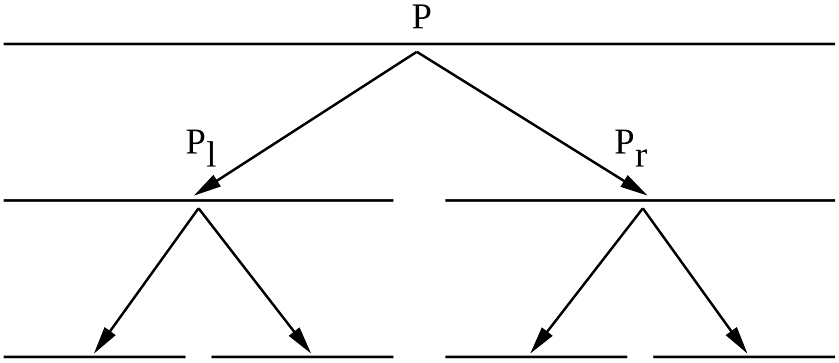


Figure 12.4: Binary Tree (subdivision of a straight line segment)

This way, we can find all the interaction forces between A and B in $O(n+m)$ time. The force calculations between one m particle will computed separately using a recursive construction. This observation gives birth the idea of hierarchical methods.

We can also describe the method in terms of potentials. If r is much larger than both r_1 and r_2, i.e., A and B are "well-separated", then we can use the pth order multipole expansion (to be given later) to express the pth order approximation of potential due to all particles in B. Let $\Phi_B^p(x)$ denote such a multipole expansion. To (approximately) compute the potential at particles in A, we simply evaluate $\Phi_B^p()$ at each particle in A. Suppose $\Phi_B^p()$ has $g(p,d)$ terms. Using multipole expansion, we reduce the number of operations to $g(p,d)(|A|+|B|)$. The error of the multipole-expansion depends on p and the ratio $\max(r_1,r_2)/r$. We say A and B are *β-well-separated*, for a $\beta > 2$, if $\max(r_1,r_2)/r \leq 1/\beta$. As shown in [41], the error of the pth order multipole expansion is bounded by $(1/(\beta-1))^p$.

12.8 Quadtree (2D) and Octtree (3D) : Data Structures for Canonical Clustering

Hierarchical N-body methods use quadtree (for 2D) and octtree (for 3D) to generate a canonical set of boxes to define clusters. The number of boxes is typically linear in the number of particles, i.e., $O(n)$.

Quadtrees and octtrees provide a way of hierarchically decomposing two dimensional and three dimensional space. Consider first the one dimensional example of a straight line segment. One way to introduce clusters is to recursively divide the line as shown in Figure 12.4.

This results in a binary tree[2].

In two dimensions, a *box* decomposition is used to partition the space (Figure 12.5). Note that a box may be regarded as a "product" of two intervals. Each partition has at most one particle in it.

A *quadtree* [83] is a recursive partition of a region of the plane into axis-aligned squares. One square, the *root*, covers the entire set of particles. It is often chosen to be the smallest (up to a constant factor) square that contains all particles. A square can be divided into four *child* squares, by splitting it with horizontal and vertical line segments through its center. The collection of squares

[2]A tree is a graph with a single *root* node and a number of subordinate nodes called *leaves* or *children*. In a binary tree, every node has at most two children.

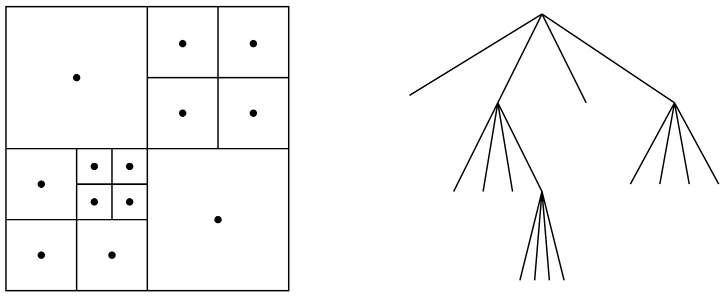


Figure 12.5: Quadtree

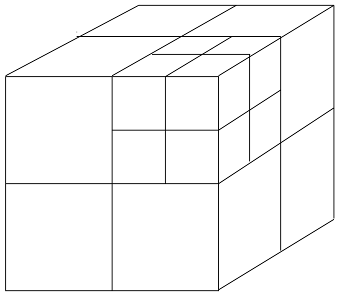


Figure 12.6: Octtree

then forms a tree, with smaller squares at lower levels of the tree. The recursive decomposition is often adaptive to the local geometry. The most commonly used termination condition is: the division stops when a box contains less than some constant (typically $m = 100$) number of particles (See Figure 12.5).

Octtree is the three-dimension version of quadtree. The root is a box covering the entire set of particles. Octtree are constructed by recursively and adaptively dividing a box into eight child-boxes, by splitting it with hyperplanes normal to each axes through its center (See Figure 12.6).

12.9 Barnes-Hut Method (1986)

The Barnes-Hut method uses these clustered data structures to represent the bodies in the simulation, and takes advantage of the distant-body simplification mentioned earlier to reduce computational complexity to $O(n\log(n))$.

The method of Barnes and Hut has two steps.

1. **Upward evaluation of center of mass**

   Refer to Figure 12.5 for the two dimensional case. Treating each box as a uniform cluster, the center of mass may be hierarchically computed. For example, consider the four boxes shown in Figure 12.7.

   The total mass of the system is

$$m = m_1 + m_2 + m_3 + m_4 \tag{12.5}$$

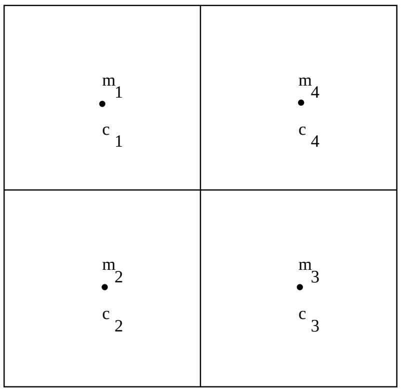


Figure 12.7: Computing the new Center of Mass

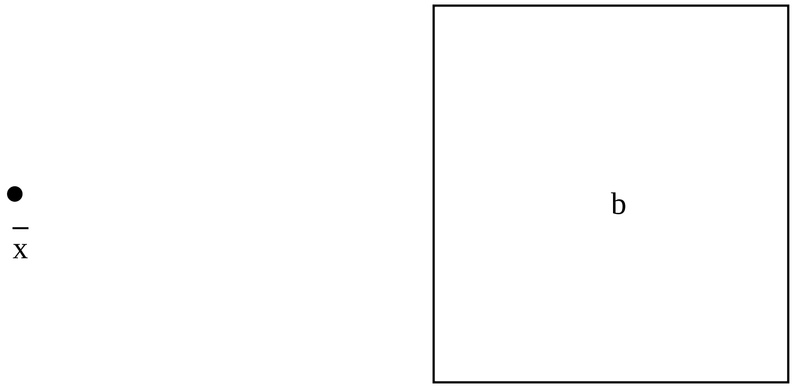


Figure 12.8: Pushing the particle down the tree

and the center of mass is given by

$$\vec{c} = \frac{m_1\vec{c_1} + m_2\vec{c_2} + m_3\vec{c_3} + m_4\vec{c_4}}{m} \tag{12.6}$$

The total time required to compute the centers of mass at all layers of the quadtree is proportional to the number of nodes, or the number of bodies, whichever is greater, or in Big-O notation, $O(n+v)$, where v is for *vertex*

This result is readily extendible to the three dimensional case.

2. **Pushing the particle down the tree**
   Consider the case of the octtree i.e. the three dimensional case. In order to evaluate the potential at a point $\vec{x}_i$, start at the top of the tree and move downwards. At each node, check whether the corresponding box, b, is well separated with respect to $\vec{x}_i$ (Figure 12.8).

   Let the force at point $\vec{x}_i$ due to the cluster b be denoted by $\vec{F}(i,b)$. This force may be calculated using the following algorithm:

   - *if b is "far" i.e. well separated from $\vec{x}_i$*, then

$$\vec{F}(\vec{x}_i) := \vec{F}(\vec{x}_i) + \frac{Gm_xM_b(\vec{x}-\vec{c_b})}{||\vec{x}_i - \vec{c_b}||^3} \qquad \text{in } \Re^3 \tag{12.7}$$

- *else if b is "close" to* $\vec{x}_i$

$$for\ k = 1\ to\ 8 \tag{12.8}$$
$$\vec{F}(\vec{x}_i) = \vec{F}(\vec{x}_i) + \vec{F}(i, child(b,k))$$
$$\tag{12.9}$$

The computational complexity of pushing the particle down the tree has the upper bound $9hn$, where h is the height of the tree and n is the number of particles. (Typically, for more or less uniformly distributed particles, $h = \log_4 n$.)

12.9.1 Approximating potentials

We now rephrase Barnes and Hut scheme in term of potentials. Let

$$\begin{aligned} m_A &= \text{total mass of particles in } \mathbf{A} \\ m_B &= \text{total mass of particles in } \mathbf{B} \\ \vec{c_A} &= \text{center of mass of particles in } \mathbf{A} \\ \vec{c_B} &= \text{center of mass of particles in } \mathbf{B} \end{aligned}$$

The potential at a point $\vec{x}$ due to the cluster $\mathbf{B}$, for example, is given by the following second order approximation:

$$\phi(\vec{x}) \approx \frac{m_B}{||\vec{x} - \vec{c_B}||}(1 + \frac{1}{\delta^2}) \qquad \text{in } \Re^3 \tag{12.10}$$

In other words, each cluster may be regarded as an individual particle when the cluster is sufficiently far away from the evaluation point $\vec{x}$.

A more advanced idea is to keep track of a higher order (Taylor expansion) approximation of the potential function induced by a cluster. Such an idea provides better tradeoff between time required and numerical precision. The following sections provide the two dimensional version of the fast multipole method developed by Greengard and Rokhlin.

The Barnes-Hut method discussed above uses the *particle-cluster* interaction between two well-separated clusters. Greengard and Rokhlin showed that the *cluster-cluster* intersection among well-separated clusters can further improve the hierarchical method. Suppose we have k clusters B_1 ..., B_k that are well-separated from a cluster A. Let $\Phi_i^p()$ be the pth order multipole expansion of B_i. Using particle-cluster interaction to approximate the far-field potential at A, we need to perform $g(p,d)|A|(|B_1| + |B_2| + ... + |B_k|)$ operations. Greengard and Rokhlin [41] showed that from $\Phi_i^p()$ we can efficiently compute a Taylor expansion $\Psi_i^p()$ centered at the centroid of A that approximates $\Phi_i^p()$. Such an operation of transforming $\Phi_i^p()$ to $\Psi_i^p()$ is called a *FLIP*. The cluster-cluster interaction first flips $\Phi_i^p()$ to $\Psi_i^p()$; we then compute $\Psi_A^p() =_{i=1}^k \Psi_i^p()$ and use $\Psi_A^p()$ to evaluate the potential at each particle in A. This reduces the number of operations to the order of

$$g(p,d)(|A| + |B_1| + |B_2| + ... + |B_k|).$$

12.10 Outline

- Introduction
- Multipole Algorithm: An Overview
- Multipole Expansion

- Taylor Expansion
- Operation No. 1 — SHIFT
- Operation No. 2 — FLIP
- Application on Quad Tree
- Expansion from 2-D to 3-D

12.11 Introduction

For N-body simulations, sometimes, it is easier to work with the (gravitational) potential rather than with the force directly. The force can then be calculated as the gradient of the potential.

In two dimensions, the potential function at z_j due to the other bodies is given by

$$\begin{aligned} \phi(z_j) &= {}^{n}_{i=1,i\neq j} q_i \log(z_j - z_i) \\ &= {}^{n}_{i=1,i\neq j} \phi_{z_i}(z_j) \end{aligned}$$

with

$$\phi_{z_i}(z) = q_i \log|z - z_i|$$

where z_1, ..., z_n the position of particles, and q_1, ..., q_n the strength of particles. The potential due to the bodies in the rest of the space is

$$\phi(z) =^{n}_{i=1} q_i \log(z - z_i)$$

which is *singular* at each potential body. (Note: actually the potential is $Re\ \phi(z)$ but we take the complex version for simplicity.)

With the Barnes and Hut scheme in term of potentials, each cluster may be regarded as an individual particle when the cluster is sufficiently far away from the evaluation point. The following sections will provide the details of the fast multipole algorithm developed by Greengard and Rokhlin.

Many people are often mystified why the Green's function is a logarithm in two dimensions, while it is $1/r$ in three dimensions. Actually there is an intuitive explanation. In d dimensions the Green's function is the integral of the force which is proportional $1/r^{d-1}$. To understand the $1/r^{d-1}$ just think that the lines of force are divided "equally" on the sphere of radius r. One might wish to imagine an d dimensional ball with small holes on the boundary filled with d dimensional water. A hose placed at the center will force water to flow out radially at the boundary in a uniform manner. If you prefer, you can imagine 1 ohm resistors arranged in a polar coordinate manner, perhaps with higher density as you move out in the radial direction. Consider the flow of current out of the circle at radius r if there is one input current source at the center.

12.12 Multipole Algorithm: An Overview

There are three important concepts in the multipole algorithm:

- function representations (multipole expansions and Taylor series)
- operators to change representations (SHIFTs and FLIPs)
- the general tree structure of the computation

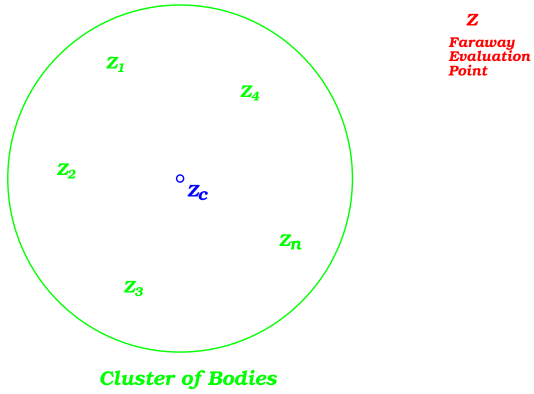


Figure 12.9: Potential of Faraway Particle due to Cluster

12.13 Multipole Expansion

The multipole algorithm flips between two point of views, or to be more precise, two representations for the potential function. One of them, which considers the cluster of bodies corresponding to many far away evaluation points, is treated in detail here. This part of the algorithm is often called the *Multipole Expansion*.

In elementary calculus, one learns about Taylor expansions for functions. This power series represents the function perfectly within the radius of convergence. A multipole expansion is also a perfectly valid representation of a function which typically converges outside *a circle rather than inside. For example, it is easy to show that*

$$\begin{aligned} \phi_{z_i}(z) &= q_i \log(z - z_i) \\ &= q_i \log(z - z_c) + {}_{k=1}^{\infty} -\frac{q_i}{k}\left(\frac{z_i - z_c}{z - z_c}\right)^k \end{aligned}$$

where z_c is any complex number. This series converges in the region $|z - z_c| > |z - z_i|$, i.e., outside of the circle containing the singularity. The formula is particularly useful if $|z - z_c| \gg |z - z_i|$, i.e., if we are far away from the singularity.

Note that

$$\begin{aligned} \phi_{z_i}(z) &= q_i \log(z - z_i) \\ &= q_i \log[(z - z_c) - (z_i - z_c)] \\ &= q_i \left[\log(z - z_c) + \log(1 - \frac{z_i - z_c}{z - z_c})\right] \end{aligned}$$

The result follows from the Taylor series expansion for $\log(1 - x)$*. The more terms in the Taylor series that are considered, the higher is the order of the approximation.*

By substituting the single potential expansion back into the main equation, we obtain the multipole expansion as following

$$\begin{aligned} \phi(z) &= {}_{i=1}^{n} \phi_{z_i}(z) \\ &= {}_{i=1}^{n} q_i \log(z - z_c) + {}_{i=1}^{n} {}_{k=1}^{\infty} q_i \left(-\frac{1}{k}\left(\frac{z_i - z_c}{z - z_c}\right)^k\right) \end{aligned}$$

$$= \quad Q\log(z - z_c) + {}_{k=1}^{\infty} a_k \left(\frac{1}{z - z_c}\right)^k$$

where

$$a_k = -{}_{i=1}^{n} \frac{q_i (z_i - z_c)^k}{k}$$

When we truncate the expansion due to the consideration of computation cost, an error is introduced into the resulting potential. Consider a p-term *expansion*

$$\phi_p(z) = Q\log(z - z_c) + {}_{k=1}^{p} a_k \frac{1}{(z - z_c)^k}$$

An error bound for this approximation is given by

$$||\phi(z) - \phi_p(z)|| \leq \frac{A}{(|\frac{z-z_c}{r}| - 1)} \left|\frac{r}{z - z_c}\right|^p$$

where r is the radius of the cluster and

$$A = {}_{i=1}^{n} |q_i|$$

This result can be shown as the following

$$\begin{aligned}
\text{Error} &= \left| {}_{k=p+1}^{\infty} a_k \frac{1}{(z - z_c)^k} \right| \\
&= \left| - {}_{k=p+1}^{\infty} {}_{i=1}^{n} \frac{q_i}{k} \left(\frac{z_i - z_c}{z - z_c}\right)^k \right| \\
&\leq {}_{k=p+1}^{\infty} {}_{i=1}^{n} |q_i| \left|\frac{r}{z - z_c}\right|^k \\
&\leq A {}_{k=p+1}^{\infty} \left|\frac{r}{z - z_c}\right|^k \\
&\leq A \frac{\left|\frac{r}{z-z_c}\right|^{p+1}}{1 - \left|\frac{r}{z-z_c}\right|} \\
&\leq \frac{A}{(|\frac{z-z_c}{r}| - 1)} \left|\frac{r}{z - z_c}\right|^p
\end{aligned}$$

At this moment, we are able to calculate the potential of each particle due to cluster of far away bodies, through multipole expansion.

12.14 Taylor Expansion

In this section, we will briefly discuss the other point of view for the multipole algorithm, which considers the cluster of evaluation points with respect to many far away bodies. It is called Taylor Expansion. For this expansion, each processor "ownes" the region of the space defined by the cluster of evaluation points, and compute the potential of the cluster through a Taylor series about the center of the cluster z_c.

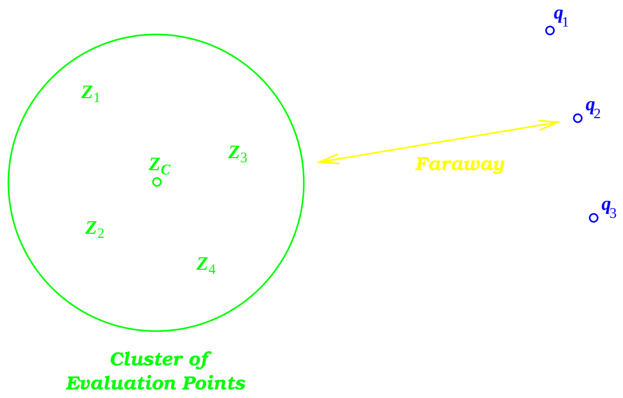


Figure 12.10: Potential of Particle Cluster

Generally, the local Taylor expansion for cluster denoted by C (with center z_c) corresponding to some body z has the form

$$\phi_{C,z_c}(z) =^{\infty}_{k=0} b_k(z - z_c)^k$$

Denote $z - z_i = (z - z_c) - (z_i - z_c) = -(z_i - z_c)(1 - \xi)$. Then for z such that $|z - z_c| < \min(z_c, C)$, we have $|\xi| < 1$ and the series $\phi_{C,z_c}(z)$ converge:

$$\begin{aligned} \phi_{C,z_c}(z) &= {}_C q_i \log(-(z_i - z_c)) +_C q_i \log(1 - \xi) \\ &= {}_C q_i \log(-(z_i - z_c)) +^{\infty}_{k=1} ({}_C q_i)\, k^{-1}(z_i - z_c)^{-k}(z - z_c)^k \\ &= b_0 +^{\infty}_{k=1} b_k(z - z_c)^k \end{aligned}$$

where formulæ for coefficients are

$$b_0 =_C q_i \log(-(z_i - z_c)) \quad \text{and} \quad b_k = k^{-1}_C q_i (z_i - z_c)^{-k} \quad k > 0.$$

Define the p-order *truncation of local expansion ϕ^p_{C,z_c} as follows*

$$\phi^p_{C,z_c}(z) =^p_{k=0} b_k(z - z_c)^k.$$

We have error bound

$$\begin{aligned} \left|\phi_{C,z_c}(z) - \phi^p_{C,z_c}(z)\right| &= \left|^{\infty}_{k=p+1} k^{-1}_C q_i \left(\frac{z - z_c}{z_i - z_c}\right)^k\right| \\ &\leq \frac{1}{p+1}{}_C |q_i|^{\infty}_{k=p+1} \left|\frac{z - z_c}{\min(z_c, C)}\right|^k = \frac{A}{(1+p)(1-c)} c^{p+1}, \end{aligned}$$

where $A =_C |q_i|$ and $c = |z - z_c| / \min(z_c, C) < 1$.

By now, we can also compute the local potential of the cluster through the Taylor expansion. During the process of deriving the above expansions, it is easy to see that

- *Both expansions are singular at the position of any body;*
- *Multipole expansion is valid outside the cluster under consideration;*
- *Taylor expansion converges within the space defined the cluster.*

At this point, we have finished the basic concepts involved in the multipole algorithm. Next, we will begin to consider some of the operations that could be performed on and between the expansions.

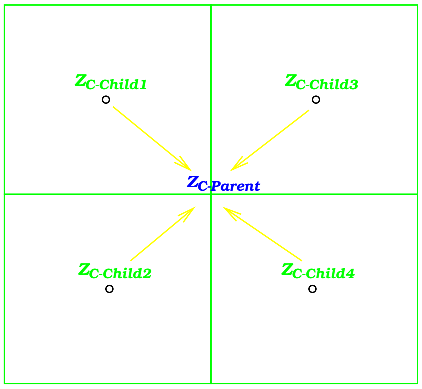


Figure 12.11: SHIFT the Center of Reference

12.15 Operation No.1 — SHIFT

Sometimes, we need to change the location of the center of the reference for the expansion series, either the multipole expansion or the local Taylor expansion. To accomplish this goal, we will perform the SHIFT operation on the expansion series.

For the multipole expansion, consider some far away particle with position z *such that both series* ϕ_{z_0} *and* ϕ_{z_1} *, corresponding to different center of reference* z_0 *and* z_1*, converge:* $|z-z_0| > \max(z_0, C)$ *and* $|z - z_1| > \max(z_1, C)$*. Note that*

$$z - z_0 = (z - z_1) - (z_0 - z_1) = (z - z_1)\left(1 - \frac{z_0 - z_1}{z - z_1}\right) = (z - z_1)(1 - \xi)$$

for appropriate ξ *and if we also assume* z *sufficiently large to have* $|\xi| < 1$*, we get identity*

$$(1 - \xi)^{-k} = \left(\sum_{l=0}^{\infty} \xi^l\right)^k = \sum_{l=0}^{\infty} \binom{k+l-1}{l} \xi^l.$$

Now, we can express the SHIFT operation for multipole expansion as

$$\begin{aligned}
\phi_{z_1}(z) &= SHIFT\,(\phi_{z_0}(z), z_0 \Rightarrow z_1) \\
&= SHIFT\left(a_0 \log(z - z_0) + \sum_{k=1}^{\infty} a_k (z - z_0)^{-k}, z_0 \Rightarrow z_1\right) \\
&= a_0 \log(z - z_1) + a_0 \log(1 - \xi) + \sum_{k=1}^{\infty} a_k (1 - \xi)^{-k} (z - z_1)^{-k} \\
&= a_0 \log(z - z_1) - a_0 \sum_{k=1}^{\infty} k^{-1} \xi^k + \sum_{k=1}^{\infty} \sum_{l=1}^{\infty} a_k \binom{k+l-1}{l} \xi^l (z - z_1)^{-k} \\
&= a_0 \log(z - z_1) + \sum_{l=1}^{\infty} \left(\sum_{k=1}^{l} a_k (z_0 - z_1)^{l-k} \binom{l-1}{k-1} - a_0 l^{-1} (z_0 - z_1)^l\right) (z - z_1)^{-l}
\end{aligned}$$

We can represent $\phi_{z_1}(z)$ *as a sequence of its coefficients* a'_k*:*

$$a'_0 = a_0 \quad \text{and} \quad a'_l = \sum_{k=1}^{l} a_k (z_0 - z_1)^{l-k} \binom{l-1}{k-1} - a_0 l^{-1} (z_0 - z_1)^l \quad l > 0.$$

Note that a'_l *depends only on* $a_0, a_1, \ldots, a_l$ *and not on the higher coefficients. It shows that given* $\phi^p_{z_0}$ *we can compute* $\phi^p_{z_1}$ *exactly, that is without any further error! In other words, operators*

SHIFT and truncation commute on multipolar expansions:

$$SHIFT(\phi^p_{z_0}, z_0 \Rightarrow z_1) = \phi^p_{z_1}.$$

Similarly, we can obtain the SHIFT operation for the local Taylor expansion, by extending the operator on the domain of local expansion, so that $SHIFT(\phi_{C,z_0}, z_0 \Rightarrow z_1)$ *produces* ϕ_{C,z_1}. *Both series converges for* z *such that* $|z - z_0| < \min(z_0, C)$, $|z - z_1| < \min(z_1, C)$. *Then*

$$\begin{aligned}
\phi_{C,z_1}(z) &= SHIFT\left(\phi_{C,z_0}(z), z_0 \Rightarrow z_1\right) \\
&= \sum_{k=0}^{\infty} b_k((z - z_1) - (z_0 - z_1))^k \\
&= \sum_{k=0}^{\infty} b_k \sum_{l=0}^{\infty} (-1)^{k-l} \binom{k}{l} (z_0 - z_1)^{k-l} (z - z_1)^l \\
&= \sum_{l=0}^{\infty} \left(\sum_{k=l}^{\infty} b_k (-1)^{k-l} \binom{k}{l} (z_0 - z_1)^{k-l} \right) (z - z_1)^l
\end{aligned}$$

Therefore, formula for transformation of coefficients b_k *of* ϕ_{C,z_0} *to* b'_l *of* ϕ_{C,z_1} *are*

$$b'_l = \sum_{k=l}^{\infty} a_k (-1)^{k-l} \binom{k}{l} (z_0 - z_1)^{k-l}.$$

Notice that in this case, b'_l *depends only on the higher coefficients, which means knowledge of the coefficients* $b_0, b_1, \ldots, b_p$ *from the truncated local expansion in* z_0 *does* not *suffice to recover the coefficients* $b'_0, b'_1, \ldots b'_p$ *at another point* z_1. *We do incur an error by the SHIFT operation applied to truncated local expansion:*

$$\begin{aligned}
\left| SHIFT(\phi^p_{C,z_0}, z_0 \Rightarrow z_1) - \phi^p_{C,z_1} \right| &= \left| \sum_{l=0}^{\infty} \left(\sum_{k=p+1}^{\infty} b_k (-1)^{k-l} (z_0 - z_1)^{k-l} \right) (z - z_1)^l \right| \\
&\leq \left| \sum_{k=p+1}^{\infty} b_k (z_1 - z_0)^k \right| \left| \sum_{l=0}^{\infty} \left(\frac{z - z_1}{z_1 - z_0} \right)^l \right| \\
&= \left| \sum_{k=p+1}^{\infty} k^{-1}_{C} q_i \left(\frac{z_1 - z_0}{z_i - z_0} \right)^k \right| \left| \sum_{l=0}^{\infty} \left(\frac{z - z_1}{z_0 - z_1} \right)^l \right| \\
&\leq \frac{A}{(p+1)(1-c)(1-D)} c^{p+1},
\end{aligned}$$

where $A = \sum_C |q_i|$. $c = |z_1 - z_0| / \min(z_0, C)$ *and* $D = |z - z_1| / |z_0 - z_1|$.

At this moment, we have obtained all the information needed to perform the SHIFT operation for both multipole expansion and local Taylor expansion. Next, we will consider the operation which can transform multipole expansion to local Taylor expansion.

12.16 Operation No.2 — FLIP

At this section, we will introduce the more powerful operation in multipole algorithm, namely the FLIP operation. For now, we will consider only the transformation in the direction from the multipole expansion $\phi_{z_0}(z)$ *to the local Taylor expansion* $\phi_{C,z_1}(z)$, *denoted by*

$$FLIP(\phi_{z_0}, z_0 \Rightarrow z_1) = \phi_{C,z_1}$$

For $|z - z_0| > \max(z_0, C)$ *and* $|z - z_1| < \min(z_1, C)$ *both series converge. Note that*

$$z - z_0 = -(z_0 - z_1)(1 - \frac{z - z_1}{z_0 - z_1}) = -(z_0 - z_1)(1 - \xi)$$

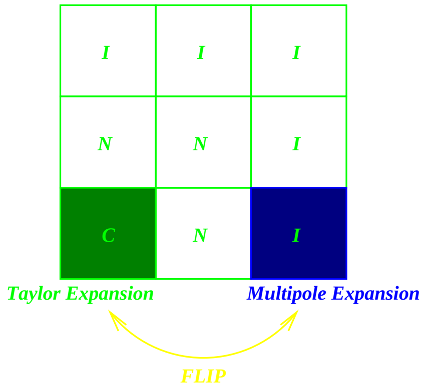


Figure 12.12: FLIP from Multipole to Taylor Expansion

and assume also $|\xi| < 1$. *Then,*

$$\begin{aligned}
\phi_{z_0}(z) &= a_0 \log(z - z_0) + \sum_{k=1}^{\infty} a_k (z - z_0)^{-k} \\
&= a_0 \log(-(z_0 - z_1)) + a_0 \log(1 - \xi) + \sum_{k=1}^{\infty} a_k (-1)^k (z_0 - z_1)^{-k} (1 - \xi)^{-k} \\
&= a_0 \log(-(z_0 - z_1)) + \sum_{l=1}^{\infty} - a_0 l^{-1} \xi^l + \sum_{k=1}^{\infty} (-1)^k a_k (z_0 - z_1)^{-k} \sum_{l=0}^{\infty} \binom{k+l-1}{l} \xi^l \\
&= \left(a_0 \log(-(z_0 - z_1)) + \sum_{k=1}^{\infty} (-1)^k a_k (z_0 - z_1)^{-k} \right) + \\
&\quad \sum_{l=1}^{\infty} \left(a_0 l^{-1} (z_0 - z_1)^{-l} + \sum_{k=1}^{\infty} (-1)^k a_k \binom{k+l-1}{l} (z_0 - z_1)^{-(k+l)} \right) (z - z_1)^l .
\end{aligned}$$

Therefore coefficients a_k *of* ϕ_{z_0} *transform to coefficients* b_l *of* ϕ_{C,z_1} *by the formula*

$$\begin{aligned}
b_0 &= a_0 \log(-(z_0 - z_1)) + \sum_{k=1}^{\infty} (-1)^k a_k (z_0 - z_1)^{-k} \\
b_l &= a_0 l^{-1} (z_0 - z_1)^{-l} + \sum_{k=1}^{\infty} (-1)^k a_k \binom{k+l-1}{l} (z_0 - z_1)^{-(k+l)} \quad l > 0
\end{aligned}$$

Note that FLIP does not *commute with truncation since one has to know all coefficients* $a_0, a_1, \ldots$ *to compute* $b_0, b_1, \ldots, b_p$ *exactly. For more information on the error in case of truncation, see Greengard and Rokhlin (1987).*

12.17 Application on Quad Tree

In this section, we will go through the application of multipole algorithm on quad tree in detail. During the process, we will also look into the two different operations $SHIFT$ *and* $FLIP$*, and gain some experience on how to use them in real situations.*

We will start at the lowest level h *of the tree. For every node of the tree, it computes the multipole expansion coefficients for the bodies inside, with origin located at the center of the cell. Next, it will shift all of the four centers for the children cells into the center of the parent node, which is at the* $h-1$ *level, through the SHIFT operation for the multipole expansion. Adding up the coefficients from the four shifted expansion series, the multipole expansion of the whole parent node is obtained. And this* $SHIFT$ *and* ADD *process will continue upward for every level of the*

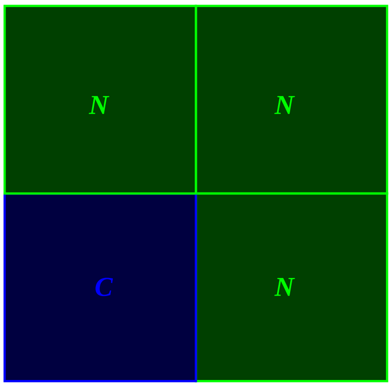


Figure 12.13: First Level of Quad Tree

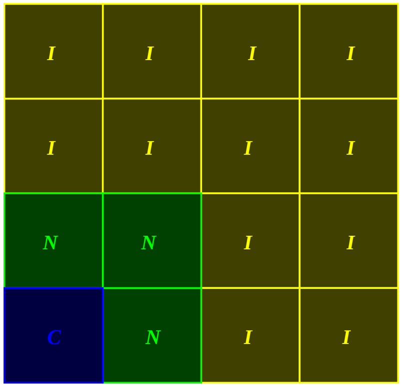


Figure 12.14: Second Level of Quad Tree

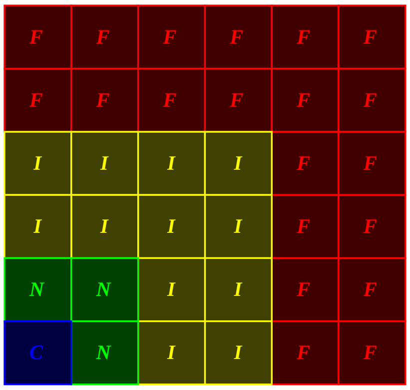


Figure 12.15: Third Level of Quad Tree

tree, until the multipole expansion coefficients for each node of the entire tree are stored within that node. The computational complexity for this part is $O(N)$.

Before we go to the next step, some terms have to be defined first.

- *NEIGHBOR — a neighbor N to a cell C is defined as any cell which shares either an edge or a corner with C*
- *INTERACTIVE — an interactive cell I to a cell C is defined as any cell whose parent is a neighbor to parent of C, excluding those which are neighbors to C*
- *FARAWAY — a faraway cell F to a cell C is defined as any cell which is neither a neighbor nor an interactive to C*

Now, we start at the top level of the tree. For each cell C, FLIP the multipole expansion for the interactive cells and combine the resulting local Taylor expansions into one expansion series. After all of the FLIP and COMBINE operations are done, SHIFT the local Taylor expansion from the node in this level to its four children in the next lower level, so that the information is conserved from parent to child. Then go down to the next lower level where the children are. To all of the cells at this level, the faraway field is done (which is the interactive zone at the parent level). So we will concentrate on the interactive zone at this level. Repeat the FLIP operation to all of the interactive cells and add the flipped multipole expansion to the Taylor expansion shifted from parent node. Then repeat the COMBINE and SHIFT operations as before. This entire process will continue from the top level downward until the lowest level of the tree. In the end, add them together when the cells are close enough.

12.18 Expansion from 2-D to 3-D

For 2-D N-body simulation, the potential function is given as

$$\phi(z_j) =_{i=1}^{n} q_i \log(z_j - z_i)$$

where $z_1, \ldots, z_n$ the position of particles, and $q_1, \ldots, q_n$ the strength of particles. The corresponding multipole expansion for the cluster centered at z_c is

$$\phi_{z_c}(z) = a_0 \log(z - z_c) +_{k=1}^{\infty} a_k \frac{1}{(z - z_c)^k}$$

The corresponding local Taylor expansion looks like

$$\phi_{C,z_c}(z) =_{k=0}^{\infty} b_k \frac{1}{(z - z_c)^k}$$

In three dimensions, the potential as well as the expansion series become much more complicated. The 3-D potential is given as

$$\Phi(x) =_{i=1}^{n} q_i \frac{1}{||x - x_i||}$$

where $x = f(r, \theta, \phi)$. The corresponding multipole expansion and local Taylor expansion as following

$$\begin{aligned} \Phi_{multipole}(x) &= {}_{n=0}^{\infty} \frac{1}{r^{n+1}} {}_{m=-n}^{n} a_n^m Y_n^m(\theta, \phi) \\ \Phi_{Taylor}(x) &= {}_{n=0}^{\infty} {}_{m=-n}^{n} r^n b_n^m Y_n^m(\theta, \phi) \end{aligned}$$

where $Y_n^m(\theta, \phi)$ is the Spherical Harmonic function. For a more detailed treatment of 3-D expansions, see Nabors and White (1991).

12.19 Parallel Implementation

In Chapter 18.2, we will discussion issues on parallel N-body implementation.

IV Numerical Linear Algebra

Chapter 13

Numerical Discretization

13.1 Mathematical Modeling and Numerical Methods

The complex mathematical models for scientific problems are typically governed by partial differential equations. Therefore, at the center of scientific computing is the numerical solution of differential equations. To "solve" PDEs on computer, we need to discretize the continuous problem and approximate it by a finite representation, usually, in the form of system of equations.

13.1.1 PDEs for Modeling

To motivate, we now list three basic partial differential equations and briefly mention the physical problems modeled by the equations. For a scalar function u (in two or three dimensions), let ∇u denote the *gradient* of u. Let $\nabla^2 u$ be the *Laplacian* of u, i.e., for two dimensional function u,

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

- **Poisson's equation** in a domain Ω with boundary Γ.

  $$-\nabla^2 u = f \qquad \text{in } \Omega$$

  The boundary value condition can be given as $u = c$ on Γ, where $c = 0$ for many problems (known as *Dirichlet condition*) or $\nabla u = g$ on Γ (known as *Neumann condition*).

  A number of problems in physics and mechanics are modeled by Poisson equation; u may represent for example a temperature, an electro-magnetic potential or the displacement of an elastic membrane fixed at the boundary under a transversal load of intensity f.

  Laplace's equation is a special case Poisson's equation with $f = 0$.

- **Navier-Stokes' equation** in a domain Ω in $\mathbb{R}^d$ with boundary Γ for incompressible flow.

  $$\frac{\partial u_i}{\partial t} + u \cdot \nabla u_i = -\frac{\partial p}{\partial x_i}\frac{1}{\rho} + \nu \nabla^2 u_i + f_i, \qquad \text{in } \Omega,\ i = 1, 2, ..., d$$
  $$\nabla \cdot u = 0 \text{in } \Omega,$$

  where $u(x,t)$ is the velocity, $p(x,t)$ is the pressure, x and t are space and time, respectively, $f(x,t)$ is the prescribed force, ρ is the fluid density, and ν is the fluid kinematic viscosity. Initial conditions are given to u and boundary conditions $u \cdot n$ and no-slip condition $u \cdot s = 0$ are given to u.

- **Schrödinger's equation** for quantum chemistry.

$$H\Psi = E\Psi,$$

where H is the *Hamiltonian operator*, given bellow, which has a Laplacian term, representing the electron kinetic energy, and a second term representing potential energy.

$$H = -_a \frac{1}{2m_a}\nabla_a^2 + \frac{1}{2}_{a,b}\frac{Q_aQ_b}{|r_a - r_b|}$$

Other important PDEs include the **Maxwell's equation** for electromagnetic, **Euler's equation** for an incompressible inviscid fluid in a domain, etc. We will discuss the application of numerical methods at the end of the tutorial.

13.1.2 Numerical Methods

There are two basic schools of numerical approaches: *equation based methods* and *particle methods.* We will cover the particle methods in Chapter **??**. In this chapter, we discuss the set of key problems in the first approach. The basic scheme of equation based numerical methods has five steps.

1. **Problem Formulation and Geometric Modeling**: describe the geometry and the boundary of a continuous domain D and the governing partial differential equations.
2. **Mesh Generation**: generate a well-shaped mesh M to approximate the domain.
3. **Numerical System Formulation**: generate a system of equations over M for the governing PDEs, for example, assemble the stiffness matrix and the right hand vector.
4. **Solution to Numerical Systems**: solve the system of equations.
5. **Adaptive Refinement**: adaptively refine the mesh based on the estimated errors (for boundary value problems) or properly move the mesh (for initial value problems), and then return to Step 3 to reduce the error or to do the next time step simulation.

We start from a continuous problem which is described by a set of PDEs on a continuous domain. We must first arrive to a discretized problem. To obtain a proper discretize problem, we need first discretize the continuous domain by a step of mesh generation. The purpose of mesh generation is to decompose the domain into a collection of simple geometric elements such as squares, boxes, or triangles.

From the discretization, we derive a finite approximation of the PDEs. There are two leading numerical formulations: Finite difference and Finite element formulations. Both methods construct a system of linear equations to approximate the PDEs.

We then need to solve the linear equations and, if necessary, adaptively refine the discretization and approximation to achieve more accurate numerical solutions.

This tutorial will discuss each of the above steps with focus on the parallel implementation. One fundamental problem that will be called upon in the parallel implementation of each of the above steps in the partitioning problem, that is how to divide a large complex problem into relatively independent ones so that we can map them on a parallel machine to achieve load balancing and to reduce communication overhead. We will discuss several practically used partitioning methods and show how to use them to support the five steps above.

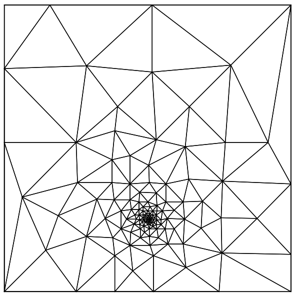


Figure 13.1: Triangulation of well-spaced point set around a singularity

13.2 Well-Shaped Meshes

Numerical methods approximate a continuous physical problem by discretizing the geometric domain into simple *cells* or *elements*. The resulting *mesh* is a graph embedded in two or three dimensions. The most versatile type of mesh is the unstructured triangular mesh in which each element is a simplex, i.e., a triangle in 2D or a tetrahedron in 3D. However, not all discretizations serve equally well. Numerical and discretization errors depend on the geometric *quality* of the mesh, which is often measured by the size and shape of the elements. A typical quality guarantee gives a lower bound on the minimum angle (e.g., greater than 30 degree). A mesh with such a quality guarantee is called a *well-shaped mesh.*

The simplest mesh is the regular Cartesian grid, which is used in finite difference methods. However, regular grids are only useful in practice for problems whose domain is simple and whose solution changes slowly. For problems with complex geometry whose solution changes rapidly, we need to use an *unstructured mesh* to reduce the problem size. For example, when modeling earthquake we want a dense discretization near the quake center and a sparse discretization in the regions with low activities. It would be waste to give regions with low activities as fine a discretization as the regions with high activities. See figure 13.1.

One important aspect that distinguishes a finite element or finite difference mesh from a regular graph is that it has two structures: the combinatorial structure and the geometric structure. In general, it can be represented by a pair (G, xyz) where G describes the combinatorial structure of the mesh and xyz gives the geometric information.

To properly approximate a continuous function, in addition to the conditions that a mesh must conform to the boundaries of the region and be fine enough, each individual element of the mesh must be *well shaped.* A common shape criterion for elements is the condition that the angles of each element are not too small, or the aspect ratio of each element is bounded [6, 33].

Several definitions of the *aspect ratio* have been used in literature. We list some of them.

1. The ratio of the longest dimension to the shortest dimension of the simplex S, denoted by $A_1(S)$. For a triangle in $\mathbb{R}^2$, it is the ratio of the longest side divided by the altitude from the longest side.

2. The ratio of the radius of the smallest containing sphere to the radius of the inscribed sphere of S, denoted by $A_2(S)$.

3. The ratio of the radius of the circumscribing sphere to the radius of the inscribed sphere of S, denoted by $A_3(S)$.

4. The ratio of the diameter to the dth root of the volume of the simplex S, denoted by $A_4(S)$, where the *diameter* of a d-simplex S is the maximum distance between any pair of points in S.

13.3 From PDEs to Systems of Linear Equations

13.3.1 Finite Difference Approximations

Finite difference uses pointwise approximation to construct a linear system $AU = F$ for PDEs $Lu = f$. In its simplest form, finite difference uses a regular grid to discretize a continuous physical domain and replace derivatives by difference quotients.

Let us first consider the two-point boundary value problem:

$$\begin{aligned} -u(x)'' = f(x) \qquad & 0 < x < 1 \\ u(0) = u(1) = 0 & \end{aligned}$$

The finite difference method discretizes the domain $[0,1]$ into n intervals by introducing the grid points $x_j = jh$, where $h = 1/n$. At each $n-1$ internal grid points, we approximate $u''(x_i)$ by $(u(x_{i-1}) - 2u(x_i) + u(x_{i+1}))/h^2$. Let $F = (f_1, ..., f_{n-1})$ and $U = (u_1, ..., u_{n-1})$, where $f_i = f(x_i)$ and $u_i = u(x_i)$. The finite difference approximation results an linear system $A^h U^h = F^h$ for the original PDE, where

$$A_h = \frac{1}{h^2}\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & 1 \\ & & & -1 & 2 \end{pmatrix}.$$

The linear system has $n-1$ variables and $n-1$ equation and its matrix A_h is symmetric positive definite.

Analogously, we can use finite difference method for approximating two dimensional PDEs. We consider the *Poisson's equation* on a Ω with boundary Γ which is given by

$$\begin{aligned} -u_{xx} - u_{yy} = f(x,y) \qquad & (x,y) \in \Omega \\ u(x,y) = 0 \qquad & (x,y) \in \Gamma. \end{aligned}$$

We will illustrate the basic formulation on the unit square, i.e., $\Omega = [0,1] \times [0,1]$.

The finite difference discretizes the domain Ω by a regular $n \times n$ grid. So each grid point is of format (x_i, y_j) where $x_i = ih$ and $y_j = jh$, and where $h = 1/n$. Replacing $-u_{xx} - u_{yy} = f(x,y)$ by finite difference, we have obtain $(n-1)^2$ linear equations of form

$$\frac{-u_{i-1,j} + 2u_{i,j} - u_{i+1,j}}{h^2} + \frac{-u_{i,j-1} + 2u_{i,j} - u_{i,j+1}}{h^2} = f_{i,j}$$
$$u_{i,j} = 0 \text{ if } i = 0 \text{ or } i = n \text{ or } j = 0 \text{ or } j = n, \text{ and } 0 < i < n,\ 0 < j < n.$$

If we order the vertices by the lexicographic ordering of their coordinates, we can express the above linear system in form of $A_{hh} U_{hh} = F_{hh}$, where

$$A_{hh} = \frac{1}{h^2}\begin{pmatrix} A_h & -I & & & \\ -I & A_h & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & A_h & I \\ & & & -I & A_h \end{pmatrix},$$

where A_h is the one dimensional finite difference matrix with space h.

Computationally we need to solve the linear system described above. Finite difference has also be extended to unstructured discretization. Mathematically, there are two key steps in the numerical analysis of the finite difference approximation:

1. Use a Taylor series expansion to bound the discretization error.

2. Show that the discretization solution U_{hh} is globally stable, that is U_{hh} depends continuously on F_{hh} as the mesh space h approach to zero.

The pointwise discretization error $E_{i,j}^{hh} = u_{x_i,x_j} - u_{i,j}^{hh}$, where $u_{i,j}$ is the exact solution of the finite difference linear system. We can show that the finite difference is a second order approximation in the sense that norm of the the discretization error is bounded as

$$||E^{hh}|| \leq Ch^2, \text{ for a constant } C \text{ independent of } h.$$

13.3.2 Finite Element Approximations

As we have seen, finite difference methods approximate PDEs by pointwise approximation. This is one of the basic approaches to make a continuous problem discrete. The other way is choose a finite number of functions $T_1, ..., T_n$, called *basis functions*. We approximate the continuous function u as a linear combination of T_i's, that is, find the set of coefficients u_i $(1 \leq i \leq n)$ so that the function $_{i=1}^{n} = u_i T_i$ is as close as to the continuous function u as possible.

Clearly, the choice of the basis functions is very important to the second approach. If these functions are sins and cosines, then we approximate the continuous function by finite Fourier series. *Finite element methods* use piecewise polynomials (or even linear functions) as basis functions. It starts with a discretization of a physical domain by a mesh (structured or unstructured). Then it generates a polynomial (or linear) basis function for each mesh point, which takes value zero outside the mesh elements that are incident to the mesh points. Therefore, we can choose basis functions that fit the geometry of the problem. The choice of polynomials or linear basis functions is to ensure that such basis functions can be generated automatically by computer from the mesh. The decision that a basis function is zero outside the neighboring elements is to reduce the complexity of the linear system generated by the finite element method, so that we only need to solve a sparse linear system as in the finite difference methods.

Galerkin Variational Formulation

To illustrate the basic idea of using basis functions, we first consider the two-point boundary value problem discussed in the previous subsection. We can show that the two-point boundary value problem is equivalent to the following variational form, known as *weak Galerkin form*: *Find an u with $u(0) = u(1) = 0$ such that $\int_0^1 -u''(x)v(x)dx = \int_0^1 f(x)v(x)dx$, for all v that satisfies $v(0) = v(1) = 0$.*

By integration by part, we have $\int_0^1 u''(x)v(x)dx = \int_0^1 u'(x)v'(x)dx$. Let $(u, v) = \int_0^1 u(x)v(x)dx$. So we are looking for a function u such that $(u', v') = (f, v)$ for all v that satisfies $v(0) = v(1) = 0$.

The basic idea of finite element formulation is to construct a set of basis functions $T_1, ..., T_n$ and find the solution to the following set of equations:

$$((_{i=1}^{n} u_i T_i)', (T_j)') = (f, T_j) \quad 1 \leq j \leq n. \tag{13.1}$$

The approximation to the continuous problem is then is given as $u =_{i=1}^{n} u_i T_i$. Clearly, given $T_1, ..., T_n$, equations (13.1) yield a linear system $Au = b$ in variables $u_1, ..., u_n$. We call A the *stiffness matrix* and b the right hand vector or *load vector*.

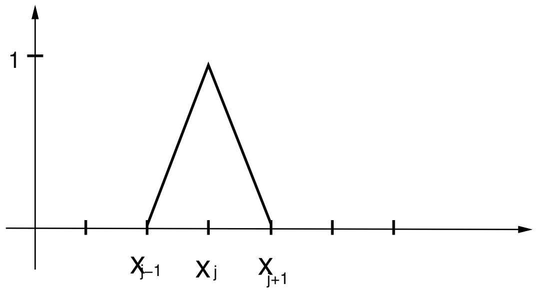


Figure 13.2: A basis function for 1D finite element

The Choice of Basis Functions

The choice of the basis functions is essential in finite element formulation. The set of basis functions need to satisfy the following conditions.

1. *Small Discretization Error*: The solution $u =_{i=1}^{n} u_i T_i$ for Equations (13.1) approximates the solution to the PDEs closely.

2. *Efficient Solution*: The resulting linear system $Ax = b$ can be solved efficiently.

Similar to the finite difference method, the finite element method first discretizes the domain. In its simplest format, it generates a linear function T_i for each mesh point p_i so that the function T_i takes value 1 at the mesh point and 0 at all other mesh points. T_i is linear in the elements that adjacent to the mesh point p_i.

We first illustrate the basic idea by the two-point boundary value problem. In one dimension, we discretize the interval $[0, 1]$ into a collection of subintervals. The basis function at a mesh point is the continuous piecewise linear function as shown in the following Figure 13.2

The final solution from the finite element formulation is also a piecewise linear function.

It is important to notice that the stiffness matrix so generated is very sparse. For one dimensional problem, it is a triangular matrix. The basis function of a mesh point only overlap with the basis functions its neighboring mesh points. In the special case of the uniform discretization of the interval, the stiffness matrix is the same as the matrix generated by the finite difference method. The stiffness matrix over a two dimension regular grid will be different from the matrix generated by the finite difference method.

Two dimensional finite element formulation is similar to its one dimensional formulation. We use the *Poisson's equation* on a domain Ω to illustrate. Recall that the Poisson's equation one Ω with boundary Γ which is given by

$$\begin{aligned} -u_{xx} - u_{yy} = f(x,y) \qquad & (x,y) \in \Omega \\ u(x,y) = 0 \qquad & (x,y) \in \Gamma. \end{aligned}$$

The finite element method constructs a mesh M of the domain Ω and use M to generate the basis and trial function.

The Galerkin variational formulation of the Poisson equation become: *Find an u which is equal to zero on Γ such that $\int\int_\Omega -(u_x x(x,y) + u_y y(x,y))v(x,y)dxdy = \int\int_\Omega f(x,y)v(x,y)dxdy$, for all v on Ω that satisfies the condition that v is equal to zero on Γ.*

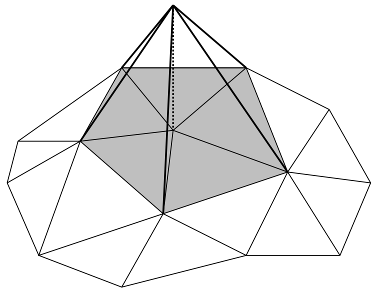


Figure 13.3: A basis function for 2D finite element

Integrating by part, we have

$$\int\int_{\Omega}(u_x x(x,y) + u_y y(x,y))v(x,y)dxdy = \int\int_{\Omega}(u_x(x,y)v_x(x,y) + u_y(x,y)v_y(x,y))dxdy.$$

So the formulation is symmetric with respect to both u and v. For each mesh point p_i of M we construct a basis function f_i which is a piecewise linear function that takes value 1 at p_i and 0 in all other mesh points. The function f_i in nonzero only in the elements that incidents to p_i. See Figure 13.3 for illustration.

Again, we assume $u =_{i=1}^{n} u_i T_i$ and use $\{T_1, ..., T_n\}$ as trial functions to generate n linear equations as following.

$$\int\int_{\Omega}{}_{i=1}^{n} u_i((T_i)_x(x,y)(T_j)_x(x,y) + (T_i)_y(x,y)(T_j)_y(x,y))dxdy \quad 1 \le j \le n.$$

From the integration formulation, we obtain a linear system $Au = b$ where A is the stiffness matrix. Notice that $A_{i,j}$ is nonzero iff p_i and p_j share an element. Therefore if we use the regular two dimensional grid as the mesh, the stiffness matrix is different from the matrix generated by the finite difference method in that each mesh point is connected to its eight nearest mesh points instead of the four as the in the finite difference method.

The finite element basis functions are more flexible than the pointwise difference formulae. For PDEs with complex geometry domain, the finite element approximation enables us to design a mesh that conform to the geometry so that we can approximate the continuous problem with smaller number of mesh points and hence with smaller number of linear equations. The Finite element method will be more important for 3D simulation where the scale of simulation is normally determined by the memory size and I/O capacity of a parallel machines.

We refer interested read to the Book of Johnson and Strang and Fix for more detailed discussion.

Chapter 14

Dense Linear Algebra

14.1 Dense Matrices

We next look at *dense* linear systems of the form $\mathbf{Ax} = \mathbf{b}$. Here A is a given $n \times n$ matrix and b is a given n-vector; we need to solve for the unknown n-vector x. We shall assume that A is a nonsingular matrix, so that for every b there is a unique solution $x = A^{-1}b$. Before solving dense linear algebra problems, we should define the terms sparse, dense, and structured.

Definition. (Wilkinson) A *sparse* matrix is a matrix with enough zeros that it is worth taking advantage of them.

Definition. A *structured* matrix has enough structure that it is worthwhile to use it.

For example, a Toeplitz Matrix is defined by $2n$ parameters. All entries on a diagonal are the same:

$$ToeplitzMatrix = \begin{bmatrix} 1 & 4 & & & & \\ 2 & 1 & 4 & & & \\ & 2 & 1 & 4 & & \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & 2 & 1 & 4 \\ & & & & 2 & 1 \end{bmatrix}$$

Is it worth taking advantage of this? If $n = 50$, hardly. The standard $O(n^3)$ algorithm for ordinary matrices can solve $Tx = b$, ignoring its structure, in under one one-hundredth of one second on a workstation. On the other hand, for large n, it pays to use one of the "fast" algorithms that run in time $O(n^2)$ or even $O(n \log^2 n)$.

Definition. A *dense* matrix is neither sparse nor structured.

14.2 Applications

There are not many applications for large dense linear algebra routines, perhaps due to the law of nature below. However, it would be wrong to conclude that the world does not need large linear algebra libraries. Medium sized problems are most easily solved with these libraries, and the first pass at larger problems are best done with the libraries. Dense methods are the easiest to use, reliable, predictable, easiest to write, and work best for small to medium problems.

Year	Size of Dense System	Machine
1950's	≈ 100	
1991	55,296	
1992	75,264	Intel
1993	75,264	Intel
1994	76,800	CM
1995	128,600	Intel
1996	128,600	Intel

Table 14.1: Largest Dense Matrices Solved

For large problems, it is not clear whether dense methods are best, but other approaches often require far more work.

- "Law of Nature": Nature does not throw n^2 numbers at us haphazardly, therefore there are few dense matrix problems. Some believe that there are no real problems that will turn up n^2 numbers to populate the $n \times n$ matrix without exhibiting some form of underlying structure. This implies that we should seek methods to identify the structure underlying the matrix. This becomes particularly important when the size of the system becomes large.

14.3 Records

Table 14.1 shows the largest dense matrices solved. Problems that warrant such huge systems to be solved are typically things like the Stealth bomber and large Boundary Element codes[1]. Another application for large dense problems arise in the "methods of moments", electro-magnetic calculations used by the military.

It is important to understand that space considerations, *not* processor speeds, are what bound the ability to tackle such large systems. Memory is the bottleneck in solving these large dense systems. Only a tiny portion of the matrix can be stored inside the computer at any one time. The record setter of size $n = 128,600$ requires $(2/3)n^3 = 1.4 \times 10^{15}$ arithmetic operations (or four times that many if it is a complex matrix) for its solution using Gaussian elimination. On a fast uniprocessor workstation today, that would take ten million seconds, about 16 and a half weeks; but on a large parallel machine, running at 1000 times this speed, the time to solve it is only 2.7 hours. The storage requirement is $8n^2 = 1.3 \times 10^13$ bytes, however. Can we afford this much main memory? If the price is even as low as \$10 per megabyte (low in 1996) it would cost \$ 130 million for enough memory for the matrix. This is a few times more than the price tag of the largest parallel machines. Hence, "out-of-core" methods are required for these giant systems.

14.4 Algorithms, and mapping matrices to processors

There is a simple minded view of parallel dense matrix computation that is based on these assumptions:

- one matrix element per processor

[1]Typically this method involves a transformation using Greens Theorem from 3D to a dense 2D representation of the problems. This is where the large data sets are generated.

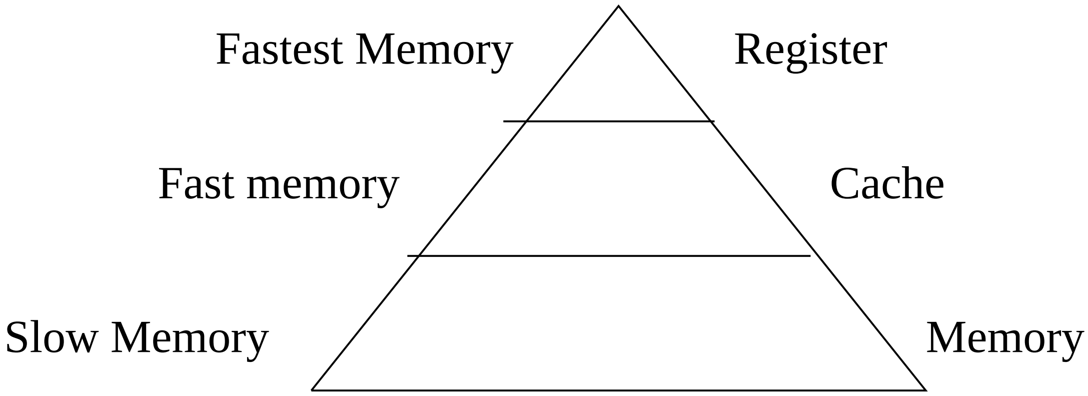


Figure 14.1: Matrix Operations on 1 processor

- a huge number (n^2, or even n^3) of processors
- communication is instantaneous

This is taught frequently in theory classes, but has no practical application. Communication cost is critical, and no one can afford n^2 processors when $n = 128,000$.[2]

In practical parallel matrix computation, it is essential to have large chunks of each matrix on each processor. There are several reasons for this. The first is simply that there are far more matrix elements than processors! Second, it is important to achieve *message vectorization*. The communications that occur should be organized into a small number of large messages, because of the high message overhead. Lastly, uniprocessor performance is heavily dependent on the nature of the local computation done on each processor.

14.5 Uniprocessor performance and the memory hierarchy

Parallel machines are built out of ordinary sequential processors. Fast microprocessors now can run far faster than the memory that supports them, and the gap is widening. The cycle time of a current microprocessor in a fast workstation is now in the $3 - 10$ nanosecond range, while DRAM memory is clocked at about 70 nanoseconds. Typically, the memory bandwidth onto the processor is close to an order of magnitude less than the bandwidth required to support the computation.

To match the bandwidths of the fast processor and the slow memory, several added layers of memory hierarchy are employed by architects. The processor has registers that are as fast as the processing units. They are connected to an on-chip cache that is nearly that fast, but is small (a few ten thousands of bytes). This is connected to an off-chip level-two cache made from fast but expensive static random access memory (SRAM) chips. Finally, main memory is built from the least cost per bit technology, dynamic RAM (DRAM). A similar caching structure supports instruction accesses.

When LINPACK was designed (the mid 1970s) these considerations were just over the horizon. Its designers used what was then an accepted model of cost: the number of arithmetic operations. Today, a more relevant metric is the number of references to memory that miss the cache and cause a cache line to be moved from main memory to a higher level of the hierarchy. To write portable software that performs well under this metric is unfortunately a much more complex task.

[2]Biological computers have this many processing elements; the human brain has on the order of 10^{11} neurons.

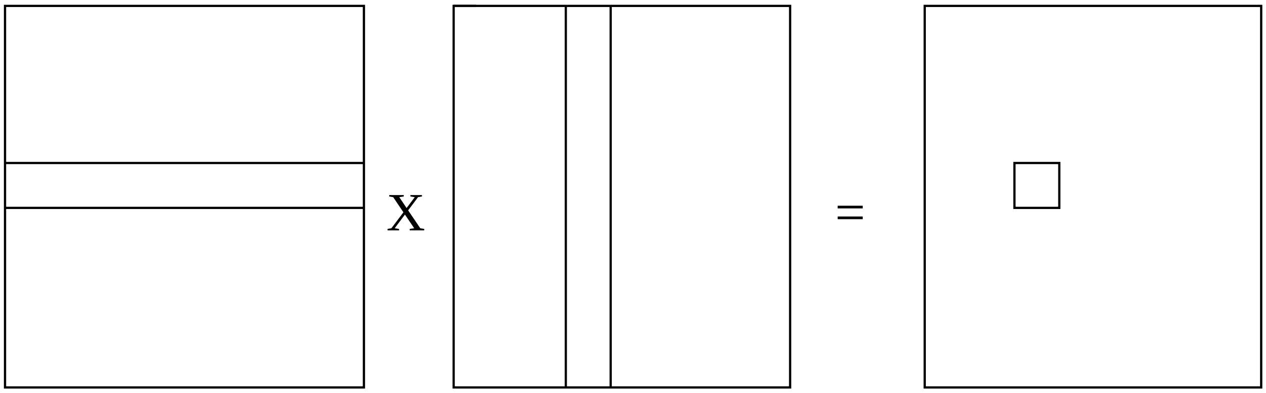


Figure 14.2: Matrix Multiply

In fact, one cannot predict how many cache misses a code will incur by examing the code. One cannot predict it by examing the machine code that the compiler generates! The behavior of real memory systems is quite complex. But, as we shall now show, the programmer can still write quite acceptable code.

(We have a bit of a paradox in that this issue does not really arise on Cray vector computers. These computers have no cache. They have no DRAM, either! The whole main memory is built of SRAM, which is expensive, and is fast enough to support the full speed of the processor. The high bandwidth memory technology raises the machine cost dramatically, and makes the programmer's job a lot simpler. When one considers the enormous cost of software, this has seemed like a reasonable tradeoff.

Why then aren't parallel machines built out of Cray's fast technology? The answer seems to be that the microprocessors used in workstations and PCs have become as fast as the vector processors. Their usual applications do pretty well with cache in the memory hierarchy, without reprogramming. Enormous investments are made in this technology, which has improved at a remarkable rate. And so, because these technologies appeal to a mass market, they have simple priced the expensive vector machines out of a large part of their market niche.)

14.6 Matrix multiply, blocked algorithms, and the BLAS

The first question to answer about an algorithm is whether it is compute bound. Matrix multiply, with its $2n^3$ operations involving $3n^2$ matrix elements, most certainly is: there is on $O(n)$ *reuse* of the data. If all the matrices fit in the cache, we get high performance. Unfortunately, we use supercomputers for big problems. The definition of "big" might well be "doesn't fit in the cache."

A typical old-style algorithm, which uses the SDOT routine from the BLAS to do the computation via inner product, is shown in Figure 14.2. This method produces disappointing performance because too many memory references are needed to do an inner product. Putting it another way, if we use this approach we will get $O(n^3)$ cache misses.

Table 14.6 shows the data reuse characteristics of several different routines in the BLAS (for Basic Linear Algebra Subprograms) library. Sequential software (LINPACK, LAPACK) for dense matrices is built on the BLAS routines.

The LAPACK designers' grand plan for high performance of portable matrix software called for manufacturers write fast BLAS, especially for the BLAS3. LAPACK codes call the BLAS. *Ergo*,

Instruction	Operations	Memory Accesses (load/stores)	Ops /Mem Ref
BLAS1: SAXPY (Single Precision **Ax** Plus **y**)	2n	3n	$\frac{2}{3}$
BLAS1: SAXPY $\alpha = \mathrm{x}\ .\ \mathrm{y}$	2n	2n	1
BLAS2: Matrix-vec $y = Ax + y$	$2n^2$	n^2	2
BLAS3: Matrix-Matrix $C = AB + C$	$2n^3$	$4n^2$	$\frac{1}{2}n$

Table 14.2: Basic Linear Algebra Subroutines (BLAS)

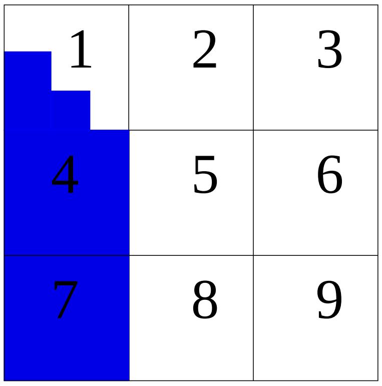


Figure 14.3: Gaussian elimination With Bad Load Balancing

LAPACK gets high performance. In reality, two things go wrong. Manufacturers dont make much of an investment in their BLAS. And LAPACK does other things, so Amdahl's law applies.

Load Balancing:

We will use Gaussian elimination to demonstrate the advantage of cyclic distribution in dense linear algebra. If we carry out Gaussian elimination on a matrix with a one-dimensional block distribution, then as the computation proceeds, processors on the left hand side of the machine become idle after all their columns of the triangular matrices L and U have been computed. This is also the case for two-dimensional block mappings. This is poor load-balancing. With cyclic mapping, we balance the load much better.

In general, there are two methods to eliminate load imbalances:

- Rearrange the data for better load balancing (costs: communication).
- Rearrange the calculation: eliminate in unusual order.

So, should we convert the data from consecutive to cyclic order and from cyclic to consecutive when we are done? The answer is "no", and the better approach is to reorganize the algorithm rather than the data. The idea behind this approach is to regard matrix indices as a set (not necessarily ordered) instead of an ordered sequence.

In general if you have to rearrange the data, maybe you can rearrange the calculation.

Lesson of Computation Distribution:

Matrix indices are a set (unordered), not a sequence (ordered). We have been taught in school to do operations in a linear order, but there is no mathematical reason to do this.

As Figure 14.4 demonstrates, we store data consecutively but do Gaussian elimination cyclicly. In particular, if the block size is 10×10, the pivots are 1, 11, 21, 31, …, 2, 22, 32, ….

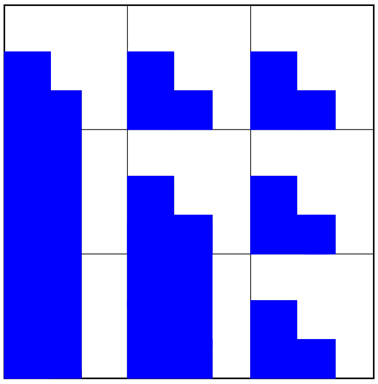


Figure 14.4: A stage in Gaussian elimination using cyclic order, where the shaded portion refers to the zeros and the unshaded refers to the non-zero elements

We can apply the above reorganized algorithm in block form, where each processor does one block at a time and cycles through.

Here we are using all of our lessons, blocking for vectorization, and rearrangement of the calculation, not the data.

14.7 Better load balancing

In reality, the load balancing achieved by the two-dimensional cyclic mapping is not all that one could desire. The problem comes from the fact that the work done by a processor that owns A_{ij} is a function of i and j, and in fact grows *quadratically* with i. Thus, the cyclic mapping tends to overload the processors with a larger first processor index, as these tend to get matrix rows that are lower and hence more expensive. A better method is to map the matrix rows to the processor rows using some heuristic method to balance the load. Indeed, this is a further extension of the moral above – the matrix row and column indices do not come from any natural ordering of the equations and unknowns of the linear system – equation 10 has no special affinity for equations 9 and 11.

14.7.1 Problems

1. For performance analysis of the Gaussian elimination algorithm, one can ignore the operations performed outside of the inner loop. Thus, the algorithm is equivalent to

```
do k = 1, n
    do j = k,n
        do i = k,n
           a(i,j) = a(i,j) - a(i,k) * a(k,j)
        enddo
    enddo
enddo
```

   The "owner" of $a(i,j)$ gets the task of the computation in the inner loop, for all $1 \leq k \leq \min(i,j)$.

Analyze the load imbalance that occurs in one-dimensional block mapping of the columns of the matrix: $n = bp$ and processor r is given the contiguous set of columns $(r-1)b+1, \ldots, rb$. (Hint: Up to low order terms, the average load per processor is $n^3/(3p)$ inner loop tasks, but the most heavily loaded processor gets half again as much to do.)

Repeat this analysis for the two-dimensional block mapping. Does this imbalance affect the scalability of the algorithm? Or does it just make a difference in the efficiency by some constant factor, as in the one-dimensional case? If so, what factor?

Finally, do an analysis for the two-dimensional cyclic mapping. Assume the $p = q^2$, and that $n = bq$ for some blocksize b. Does the cyclic method remove load imbalance completely?

Chapter 15

Sparse Linear Algebra

The solution of a linear system $Ax = b$ is one of the most important computational problems in scientific computing. As we shown in the previous section, these linear systems are often derived from a set of *differential equations*, by either finite difference or finite element formulation over a discretized mesh.

The matrix A of a discretized problem is usually very *sparse*, namely it has enough zeros that can be taken advantage of algorithmically. Sparse matrices can be divided into two classes: *structured sparse matrices* and *unstructured sparse matrices*. A structured matrix is usually generated from a structured regular grid and an unstructured matrix is usually generated from a non-uniform, unstructured grid. Therefore, sparse techniques are designed in the simplest case for structured sparse matrices and in the general case for unstructured matrices.

15.1 Cyclic Reduction for Structured Sparse Linear Systems

The simplest structured linear system is perhaps the tridiagonal system of linear equations $Ax = b$ where A is symmetric and positive definite and of form

$$A = \begin{pmatrix} b_1 & c_1 & & & \\ c_1 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & c_{n-2} & b_{n-1} & c_{n-1} \\ & & & c_{n-1} & b_n \end{pmatrix}$$

For example, the finite difference formulation of the one dimensional model problems

$$-u''(x) + \sigma u(x) = f(x), \qquad 0 < x < 1, \sigma \geq 0 \tag{15.1}$$

subject to the boundary conditions $u(0) = u(1) = 0$, on a uniform discretization of spacing h yields of a triangular linear system of $n = 1/h$ variables, where $b_i = 2 + \sigma h^2$ and $c_i = -1$ for all $1 \leq i \leq n$.

Sequentially, we can solve a triangular linear system $Ax = b$ by factor A into $A = LDL^T$, where D is a diagonal matrix with diagonal $(d_1, d_2, ..., d_n)$ and L is of form

$$L = \begin{pmatrix} 1 & 0 & & \\ e_1 & 1 & 0 & \\ & \ddots & \ddots & \ddots \\ & & e_{n-1} & 1 \end{pmatrix}.$$

The factorization can be computed by the following simple algorithm.

Algorithm Sequential Tridiagonal Solver

1. $d_1 = b_1$
2. $e_=c_1/d_1$
3. for $i = 2 : n$
   (a) $d_1 = b_i - e_{i-1}c_{i-1}$
   (b) if $i < n$ then $e_i = c_i/d_i$

The number float point operations is $3n$ upto a additive constant. With such factorization, we can then solve the tridiagonal linear system in additional $5n$ float point operations. However, this method is very reminiscent to the naive sequential algorithm for the prefix sum whose computation graph has a critical path of length $O(n)$. The cyclic reduction, developed by Golub and Hockney [?], is very similar to the parallel prefix algorithm presented in Section ?? and it reduces the length of dependency in the computational graph to the smallest possible.

The basic idea of the cyclic reduction is to first eliminate the odd numbered variables to obtain a tridiagonal linear system of $\lceil n/2 \rceil$ equations. Then we solve the smaller linear system recursively. Note that each variable appears in three equations. The elimination of the odd numbered variables gives a tridiagonal system over the even numbered variables as following:

$$c'_{2i-2}x_{2i-2} + b'_{2i}x_{2i} + c'_{2i}x_{2i+2} = f'_{2i},$$

for all $2 \leq i \leq n/2$, where

$$\begin{aligned}
c'_{2i-2} &= -(c_{2i-2}c_{2i-1}/b_{2i-1}) \\
b'_{2i} &= (b_{2i} - c^2_{2i-1}/b_{2i-1} - c^2 2i/b_{2i+1}) \\
c'_{2i} &= c_{2i}c_{2i+1}/b_{2i+1} \\
f'_{2i} &= f_{2i} - c_{2i-1}f_{2i-1}/b_{2i-1} - c_{2i}f_{2i+1}/b_{2i+1}
\end{aligned}$$

Recursively solving this smaller linear tridiagonal system, we obtain the value of x_{2i} for all $i = 1, ..., n/2$. We can then compute the value of x_{2i-1} by the simple equation:

$$x_{2i-1} = (f_{2i-1} - c_{2i-2}x_{2i-2} - c_{2i-1}x_{2i})/b_{2i-1}.$$

By simple calculation, we can show that the total number of float point operations is equal to $16n$ upto an additive constant. So the amount of total work is doubled compare with the sequential algorithm discussed. But the length of the critical path is reduced to $O(\log n)$. It is worthwhile to point out the the total work of the parallel prefix sum algorithm also double that of the sequential algorithm. Parallel computing is about the trade-off of parallel time and the total work. The discussion show that if we have n processors, then we can solve a tridiagonal linear system in $O(\log n)$ time.

When the number of processor p is much less than n, similar to prefix sum, we hybrid the cyclic reduction with the sequential factorization algorithm. We can show that the parallel float point operations is bounded by $16n(n + \log n)/p$ and the number of round of communication is bounded by $O(\log p)$. The communication pattern is the nearest neighbor.

Cyclic Reduction has been generalized to two dimensional finite difference systems where the matrix is a block tridiagonal matrix.

15.2 Sparse Direct Methods

Direct methods for solving sparse linear systems are important because of their generality and robustness. For linear systems arising in certain applications, such as linear programming and some structural engineering applications, they are the only feasible methods for numerical factorization.

15.2.1 LU Decomposition and Gaussian Elimination

The basis of direct methods for linear system is Gaussian Elimination, a process where we zero out certain entry of the original matrix in a systematically way. Assume we want to solve $Ax = b$ where A is a sparse $n \times n$ symmetric positive definite matrix. The basic idea of the direct method is to factor A into the product of triangular matrices $A = LL^T$. Such a procedure is called *Cholesky factorization.*

The first step of the Cholesky factorization is given by the following matrix fractorization:

$$A = \begin{pmatrix} d & v^T \\ v & C \end{pmatrix} = \begin{pmatrix} \sqrt{d} & 0 \\ v/\sqrt{d} & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & C - (vv^T)/d \end{pmatrix} \begin{pmatrix} \sqrt{d} & v^T/\sqrt{d} \\ 0 & I \end{pmatrix}$$

where v is $n-1 \times 1$ and C is $n-1 \times n-1$. Note that d is positive since A is positive definite. The term $C - \frac{vv^t}{d}$ is the Schur complement of A. This step is called elimination and the element d is the pivot. The above decomposition is now carried out on the Schur complement recursively. We therefore have the following algorithm for the Cholesky decomposition.

For $k = 1, 2, \ldots, n$
  $a(k,k) = \sqrt{a(k,k)}$
  $a(k+1:n, k) = \frac{a(k+1:n,k)}{a(k,k)}$
  $a(k+1:n, k+1:n) = a(k+1:n, k+1:n) - a(k+1:n, k)^T a(k+1:n, k)$
end

The entries on and below the diagonal of the resulting matrix are the entries of L. The main step in the algorithm is a rank 1 update to an $n-1 \times n-1$ block.

Notice that some fill-in may occur when we carry out the decomposition. i.e., L may be significantly less sparse than A. An important problem in direct solution to sparse linear system is to find a "good" ordering of the rows and columns of the matrix to reduce the amount of fill.

As we showed in the previous section, the matrix of the linear system generated by the finite element or finite difference formulation is associated with the graph given by the mesh. In fact, the nonzero structure of each matrix A can be represented by a graph, $G(A)$, where the rows is represented by a vertex and every nonzero element by an edge. Graph theoretically, the graph of the Schur complement after eliminating a row can be obtained from $G(A)$ by deleting the node of the row and connect its graph neighbor by a complete graph. See Figure 15.1.

The fill-in of the Cholesky decomposition can be computed as follows. First form a clique on the neighbors of node 1. Then form a clique on the neighbors of node 2 etc. The edges which are added correspond to entries of the matrix which are filled in. (Of course the filled in entry might be a zero due to cancelation but we ignore this possibility.) In our matrix the filled in entries are given by $\times$. This calculation of fill-in can be done relatively cheaply.

In the context of parallel computation, an important parameter the height of elimination tree, which is the number of parallel elimination steps need to factor with an unlimited number of processors. The *elimination tree* defined as follows from the fill-in calculation which was described above. Let $j > k$. Define $j >_L k$ if $l_{jk} \neq 0$ where l_{jk} is the (j,k) entry of L, the result of the

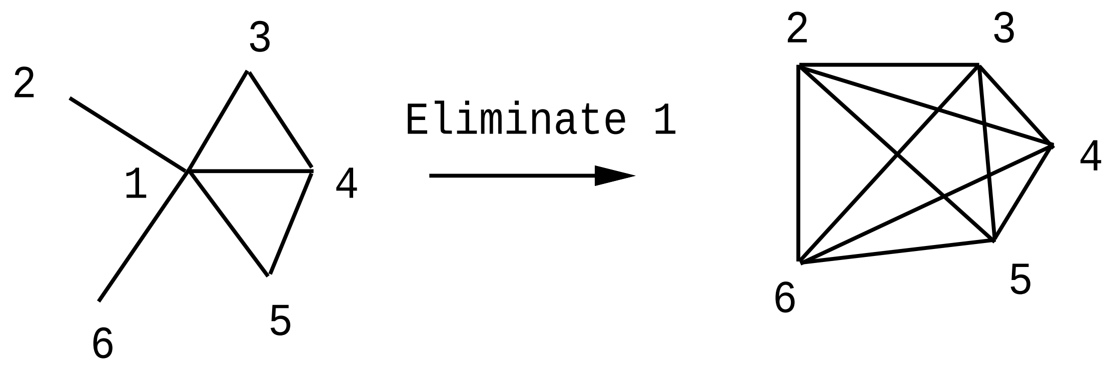


Figure 15.1: Graphical Representation of Elimination

decomposition. Let the parent of k, $p(k)$ =min$\{j : j >_L k\}$. This defines a tree since if $\beta >_L \alpha$, $\gamma >_L \alpha$ and $\gamma > \beta$ then $\gamma >_L \beta$.

The order of elimination determines both fill and elimination tree height. Unfortunately, but inevitably, finding the best ordering is NP-complete. Heuristics are used to reduce fill-in. The following lists some commonly used ones.

- Ordering by minimum degree (this is SYMMD in Matlab)
- nested dissection
- Cuthill-McKee ordering.
- reverse Cuthill-McKee (SYMRCM)
- ordering by number of non-zeros (COLPERM or COLMMD)

We now examine an ordering method called *nested dissection*, which uses vertex separators in a divide-and-conquer node ordering for sparse Gaussian elimination. Nested dissection [35, 36, 55] was originally a sequential algorithm, pivoting on a single element at a time, but it is an attractive parallel ordering as well because it produces blocks of pivots that can be eliminated independently in parallel [9, 28, 38, 57, 70].

Consider a regular finite difference grid. By dissecting the graph along the center lines (enclosed in dotted curves), the graph is split into four independent graphs, each of which can be solved in parallel.

The connections are included only at the end of the computation in an analogous way to domain decomposition discussed in earlier lectures. Figure 15.3 shows how a single domain can be split up into two roughly equal sized domains **A** and **B** which are independent and a smaller domain **C** that contains the connectivity.

One can now recursively order **A** and **B**, before finally proceeding to **C**. More generally, begin by recursively ordering at the leaf level and then continue up the tree. The question now arises as to how much fill is generated in this process. A recursion formula for the fill F generated for such a 2-dimension nested dissection algorithm is readily derived.

$$F(n) = 4F(\frac{n}{2}) + \frac{(2\sqrt{n})^2}{2} \tag{15.2}$$

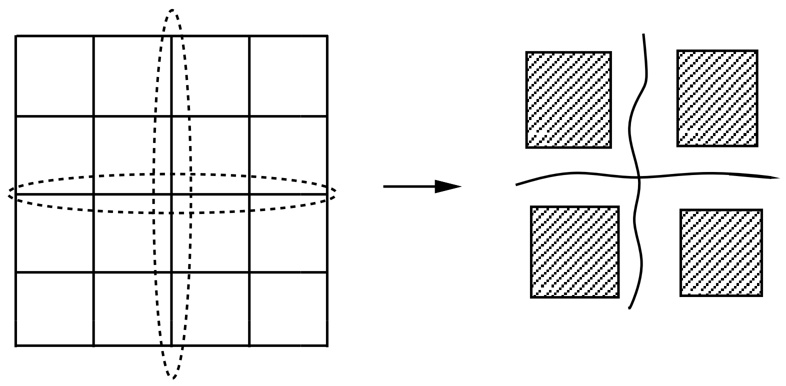


Figure 15.2: Nested Dissection

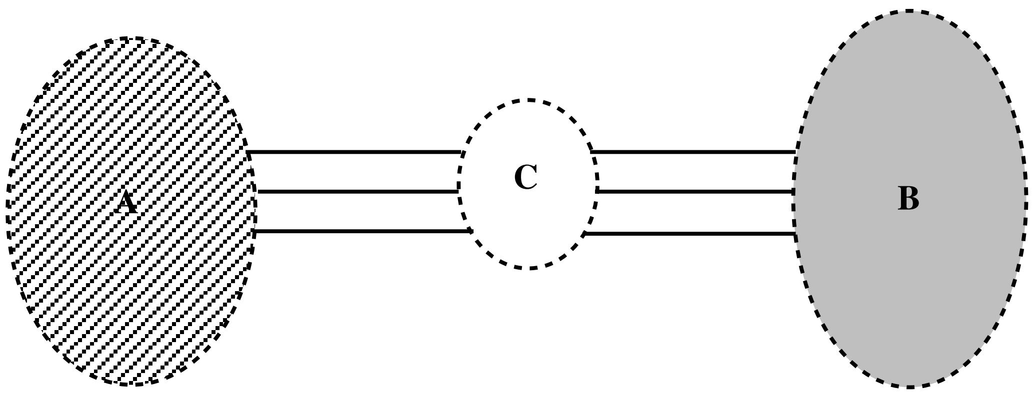


Figure 15.3: Vertex Separators

This yields upon solution

$$F(n) = 2n\log(n) \tag{15.3}$$

In an analogous manner, the elimination tree height is given by:

$$H(n) = H(\frac{n}{2}) + 2\sqrt{n} \tag{15.4}$$

$$H(n) = const \times \sqrt{n} \tag{15.5}$$

Nested dissection can be generalized to three dimensional regular grid or other classes of graphs that have small separators. We will come back to this point in the section of graph partitioning.

15.2.2 Parallel Factorization: the Multifrontal Algorithm

Nested dissection and other heuristics give the ordering. To factor in parallel, we need not only find a good ordering in parallel, but also to perform the elimination in parallel. To achieve better parallelism and scalability in elimination, a popular approach is to modify the algorithm so that we are performing a rank k update to an $n-k \times n-k$ block. The basic step will now be given by

$$A = \begin{pmatrix} D & V^T \\ V & C \end{pmatrix} = \begin{pmatrix} L_D & 0 \\ VL_D^{-T} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & C - VD^{-1}V^T \end{pmatrix} \begin{pmatrix} L_D^T & L_D^{-1}V^T \\ 0 & I \end{pmatrix}$$

where C is $n-k \times n-k$, V is $n-k \times k$ and $D = L_D L_D^T$ is $k \times k$. D can be written in this way since A is positive definite. Note that $VD^{-1}V^T = (VL_D^{-T})(L_D^{-1}V^T)$.

The elimination tree shows where there is parallelism since we can "go up the separate branches in parallel." i.e. We can update a column of the matrix using only the columns below it in the elimination tree. This leads to the multifrontal algorithm. The sequential version of the algorithm is given below. For every column j there is a block $\bar{U}_j$ (which is equivalent to $VD^{-1}V^T$).

$$\bar{U}_j = -_k \begin{pmatrix} l_{jk} \\ l_{i_1k} \\ \vdots \\ l_{i_rk} \end{pmatrix} (l_{jk}\ l_{i_1k}\ \ldots\ l_{i_rk})$$

where the sum is taken over all descendants of j in the elimination tree. $j, i_1, i_2, \ldots, i_r$ are the indices of the non-zeros in column j of the Cholesky factor.

For $j = 1, 2, \ldots, n$. Let $j, i_1, i_2, \ldots, i_r$ be the indices of the non-zeros in column j of L. Let $c_1, \ldots, c_s$ be the children of j in the elimination tree. Let $\bar{U} = U_{c_1} \updownarrow \ldots \updownarrow U_{c_s}$ where the U_i's were defined in a previous step of the algorithm. $\updownarrow$ is the extend-add operator which is best explained by example. Let

$$R = \begin{array}{c} \\ 5 \\ 8 \end{array}\!\!\begin{array}{c} \begin{matrix} 5 & 8 \end{matrix} \\ \begin{pmatrix} p & q \\ u & v \end{pmatrix} \end{array}, \quad S = \begin{array}{c} \\ 5 \\ 9 \end{array}\!\!\begin{array}{c} \begin{matrix} 5 & 9 \end{matrix} \\ \begin{pmatrix} w & x \\ y & z \end{pmatrix} \end{array}$$

(The rows of R correspond to rows 5 and 8 of the original matrix etc.) Then

$$R \updownarrow S = \begin{array}{c} \\ 5 \\ 8 \\ 9 \end{array}\!\!\begin{array}{c} \begin{matrix} 5 & 8 & 9 \end{matrix} \\ \begin{pmatrix} p+w & q & x \\ u & v & 0 \\ y & 0 & z \end{pmatrix} \end{array}$$

Define

$$F_j = \begin{pmatrix} a_{jj} & \dots & a_{ji_r} \\ \vdots & \ddots & \\ a_{i_rj} & \dots & a_{i_ri_r} \end{pmatrix} \updownarrow \bar{U}$$

(This corresponds to $C - VD^{-1}V^T$)
Now factor F_j

$$\begin{pmatrix} l_{jj} & 0 & \dots & 0 \\ l_{i_1j} & & & \\ \vdots & & I & \\ l_{i_rj} & & & \end{pmatrix} \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & U_j & \\ 0 & & & \end{pmatrix} \begin{pmatrix} l_{jj} & l_{i_1j} & \dots & l_{i_rj} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{pmatrix}$$

(Note that U_j has now been defined.)

We can use various BLAS kernels to carry out this algorithm. Recently, Kumar and Karypis have shown that direct solver can be parallelized efficiently. They have designed a parallel algorithm for factorization of sparse matrices that is more scalable than any other known algorithm for this problem. They have shown that our parallel Cholesky factorization algorithm is asymptotically as scalable as any parallel formulation of dense matrix factorization on both mesh and hypercube architectures. Furthermore, their algorithm is equally scalable for sparse matrices arising from two- and three-dimensional finite element problems.

They have also implemented and experimentally evaluated the algorithm on a 1024-processor nCUBE 2 parallel computer and a 1024-processor Cray T3D on a variety of problems. In structural engineering problems (Boeing-Harwell set) and matrices arising in linear programming (NETLIB set), the preliminary implementation is able to achieve 14 to 20 GFlops on a 1024-processor Cray T3D.

In its current form, the algorithm is applicable only to Cholesky factorization of sparse symmetric positive definite (SPD) matrices. SPD systems occur frequently in scientific applications and are the most benign in terms of ease of solution by both direct and iterative methods. However, there are many applications that involve solving large sparse linear systems which are not SPD. An efficient parallel algorithm for a direct solution to non-SPD sparse linear systems will be extremely valuable because the theory of iterative methods is far less developed for general sparse linear systems than it is for SPD systems.

15.3 Basic Iterative Methods

These methods will focus on the solution to the linear system $Ax = b$ where $A \in \mathcal{R}^{n\times n}$ and $x, b \in \mathcal{R}^n$, although the theory is equally valid for systems with complex elements.

Choose some initial guess, x_o, for the solution vector. Generate a series of solution vectors, $\{x_1, x_2, \dots, x_k\}$, through an iterative process taking advantage of previous solution vectors.

Define x^* as the true (optimal) solution vector. Each iterative solution vector is chosen such that the absolute error, $e_i = \|x^* - x_i\|$, is decreasing with each iteration for some defined norm. Define also the residual error, $r_i = \|b - Ax_i\|$, at each iteration. These error quantities are clearly related by a simple transformation through A.

$$r_i = b - Ax_i = Ax^* - Ax_i = Ae_i$$

15.3.1 Jacobi Method

Perform the matrix decomposition $A = D - L - U$ where D is some diagonal matrix, L is some strictly lower triangular matrix and U is some strictly upper triangular matrix.

Any solution satisfying $Ax = b$ is also a solution satisfying $Dx = (L + U)x + b$. This presents a straightforward iteration scheme with small computation cost at each iteration. Solving for the solution vector on the right-hand side involves inversion of a diagonal matrix. Assuming this inverse exists, the following iterative method may be used.

$$x_i = D^{-1}(L + U)\, x_{i-1} + D^{-1}b$$

This method presents some nice computational features. The inverse term involves only the diagonal matrix, D. The computational cost of computing this inverse is minimal. Additionally, this may be carried out easily in parallel since each entry in the inverse does not depend on any other entry.

15.3.2 Gauss-Seidel Method

This method is similar to Jacobi Method in that any solution satisfying $Ax = b$ is now a solution satisfying $(D - L)x = Ub$ for the $A = D - L - U$ decomposition. Assuming an inverse exists, the following iterative method may be used.

$$x_i = (D - L)^{-1}Ux_{i-1} + (D - L)^{-1}$$

This method is often stable in practice but is less easy to parallelize. The inverse term is now a lower triangular matrix which presents a bottleneck for parallel operations.

This method presents some practical improvements over the Jacobi method. Consider the computation of the j^{th} element of the solution vector x_i at the i^{th} iteration. The lower triangular nature of the inverse term demonstrates only the information of the $(j + 1)^{th}$ element through the n^{th} elements of the previous iteration solution vector x_{i-1} are used. These elements contain information not available when the j^{th} element of x_{i-1} was computed. In essence, this method updates using only the most recent information.

15.3.3 Splitting Matrix Method

The previous methods are specialized cases of Splitting Matrix algorithms. These algorithms utilize a decomposition $A = M - N$ for solving the linear system $Ax = b$. The following iterative procedure is used to compute the solution vector at the i^{th} iteration.

$$Mx_i = Nx_{i-1} + b$$

Consider the computational tradeoffs when choosing the decomposition.

- cost of computing M^{-1}
- stability and convergence rate

It is interesting the analyze convergence properties of these methods. Consider the definitions of absolute error, $e_i = x^* - x_i$, and residual error, $r_i = Ax_i - b$. An iteration using the above algorithm yields the following.

$$\begin{aligned} x_1 &= M^{-1}Nx_0 + M^{-1}b \\ &= M^{-1}(M-A)x_0 + M^{-1}b \\ &= x_0 + M^{-1}r_0 \end{aligned}$$

A similar form results from considering the absolute error.

$$\begin{aligned} x^* &= x_0 + e_0 \\ &= x_0 + A^{-1}r_0 \end{aligned}$$

This shows that the convergence of the algorithm is in some way improved if the M^{-1} term approximates A^{-1} with some accuracy. Consider the amount of change in the absolute error after this iteration.

$$\begin{aligned} e_1 &= A^{-1}r_0 - M^{-1}r_0 \\ &= e_0 - M^{-1}Ae_0 \\ &= M^{-1}Ne_0 \end{aligned}$$

Evaluating this change for a general iteration shows the error propagation.

$$e_i = \left(M^{-1}N\right)^i e_0$$

This relationship shows a bound on the error convergence. The largest eigenvalue, or spectral eigenvalue, of the matrix $M^{-1}N$ determines the rate of convergence of these methods. This analysis is similar to the solution of a general difference equation of the form $x_k = Ax_{k-1}$. In either case, the spectral radius of the matrix term must be less than 1 to ensure stability. The method will converge to 0 faster if all the eigenvalue are clustered near the origin.

15.3.4 Weighted Splitting Matrix Method

The splitting matrix algorithm may be modified by including some scalar weighting term. This scalar may be likened to the free scalar parameter used in practical implementations of Newton's method and Steepest Descent algorithms for optimization programming. Choose some scalar, w, such that $0 < w < 1$, for the following iteration.

$$\begin{aligned} x_i &= (1-w)x_0 + w\left(x_0 + M^{-1}v_0\right) \\ &= x_0 + wM^{-1}v_0 \end{aligned}$$

15.4 Red-Black Ordering for parallel Implementation

The concept of ordering seeks to separate the nodes of a given domain into subdomains. Red-black ordering is a straightforward way to achieve this. The basic concept is to alternate assigning a "color" to each node. Consider the one- and two-dimensional examples on regular grids..

The iterative procedure for these types of coloring schemes solves for variables at nodes with a certain color, then solves for variables at nodes of the other color. A linear system can be formed with a block structure corresponding to the color scheme.

$$\begin{bmatrix} \text{BLACK} & \text{MIXED} \\ \text{MIXED} & \text{RED} \end{bmatrix}$$

This method can easily be extended to include more colors. A common practice is to choose colors such that no nodes has neighbors of the same color. It is desired in such cases to minimize the number of colors so as to reduce the number of iteration steps.

15.5 Conjugate Gradient Method

The Conjugate Gradient Method is the most prominent iterative method for solving sparse symmetric positive definite linear systems. We now examine this method from parallel computing perspective. The following is a copy of a pseudocode for the conjugate gradient algorithm.
Algorithm: Conjugate Gradient

1. $\underline{x_0} = 0, \quad r_0 = b - A\underline{x_0} = b$
2. do $m = 1$, to n steps
   (a) if $m = 1$, then $p_1 = r_0$
   else
   $\beta = r_{m-1}^T r_{m-1} / r_{m-2}^T r_{m-2}$
   $p_m = r_{m-1} + \beta p_{m-1}$
   endif
   (b) $\alpha_m = r_{m-1}^T r_{m-1} / p_m^T A p_m$
   (c) $x_m = x_{m-1} + \alpha_m p_m$
   (d) $r_m = r_{m-1} - \alpha_m A p_m$

When A is symmetric positive definite, the solution of $Ax = b$ is equivalent to find a solution to the following quadratic minimization problem.

$$\min_{\underline{x}} \phi(x) = \frac{1}{2} x^T A x - x^T b.$$

In this setting, $r_0 = -\nabla \phi$ and $p_i^T A p_j = 0$, i.e., p_i^T and p_j are *conjugate* with respect to A.
How many iterations shall we perform and how to reduce the number of iterations?

Theorem 15.5.1 *Suppose the condition number is $\kappa(A) = \lambda_{max}(A)/\lambda_{min}(A)$, since A is Symmetric Positive Definite, $\forall x_0$, suppose x^* is a solution to $Ax = b$, then*

$$||x^* - x_m||_A \leq 2||x^* - x_0||_A \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^m,$$

where $||V||_A = V^T A V$

Therefore, $||e_m|| \leq 2||e_0|| \cdot (\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})^m$.
Another high order iterative method is Chebyshev iterative method. We refer interested readers to the book by Own Axelsson (Iterative Solution Methods, Cambridge University Press). Conjugate gradient method is a special Krylov subspace method. Other examples of Krylov subspace are GMRES (Generalized Minimum Residual Method) and Lanczos Methods.

15.6 Preconditioning

Preconditioning is important in reducing the number of iterations needed to converge in many iterative methods. Given a linear system $Ax = b$ a parallel preconditioner is an invertible matrix C satisfying the following:

1. The inverse C^{-1} is relatively easy to compute. More precisely, after preprocessing the matrix C, solving the linear system $Cy = b'$ is much easier than solving the system $Ax = b$. Further, there are fast parallel solvers for $Cy = b'$.

2. Iterative methods for solving the system $C^{-1}Ax = C^{-1}b$, such as, conjugate gradient[1] should converge much more quickly than they would for the system $Ax = b$.

Generally a preconditioner is intended to reduce $\kappa(A)$.

Now the question is: *how to choose a preconditioner C?* There is no definite answer to this. We list some of the popularly used preconditioning methods.

- The basic splitting matrix method and SOR can be viewed as preconditioning methods.

- **Incomplete factorization preconditioning**: the basic idea is to first choose a good "sparsity pattern" and perform factorization by Gaussian elimination. The method rejects those fill-in entries that are either small enough (relative to diagonal entries) or in position outside the sparsity pattern. In other words, we perform an approximate factorization L^*U^* and use this product as a preconditioner. One effective variant is to perform block incomplete factorization to obtain a preconditioner.

- **Subgraph preconditioning**: The basic idea is to choose a subgraph of the graph defined by the matrix of the linear system so that the linear system defined by the subgraph can be solved efficiently and the edges of the original graph can be embedded in the subgraph with small congestion and dilation, which implies small condition number of the preconditioned matrix. In other words, the subgraph can "support" the original graph.

- **Block Diagonal Preconditioning**: The observation of this method is that a matrix in many applications can be naturally partitioned in the form of a 2×2 blocks

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

  Moreover, the linear system defined by A_{11} can be solved more efficiently. Block diagonal preconditioning chooses a preconditioner with format

$$C = \begin{pmatrix} B_{11} & 0 \\ 0 & B_{22} \end{pmatrix}$$

  with the condition that B_{11} and B_{22} are symmetric and

$$\begin{aligned} \alpha_1 A_{11} \le B_{11} \le \alpha_2 A_{11} \\ \beta_1 A_{22} \le B_{22} \le \beta_2 A_{22} \end{aligned}$$

  Block diagonal preconditioning methods are often used in conjunction with domain decomposition technique. We can generalize the 2-block formula to multi-blocks, which correspond to multi-region partition in the domain decomposition,

Other preconditioning methods include Schur complement preconditioning and various heuristics of computing an approximate matrix inverses and use them as preconditioners.

[1] In general the matrix $C^{-1}A$ is not symmetric. Thus the formal analysis uses the matrix LAL^T where $C^{-1} = LL^T$ [?].

15.7 Main Issues

The most computational expensive step in conjugate gradient (and the Krylov method in general) is a sparse matrix-vector multiplication. This is true for most iterative methods. We will discuss parallel sparse matrix-vector product in the section of graph partitioning.

Numerically and algorithmically, the important problem is to efficiently construct good preconditioners.

15.8 Efficient sparse matrix algorithms

15.8.1 Scalable algorithms

By a *scalable* algorithm for a problem, we mean one that maintains efficiency bounded away from zero as the number p of processors grows and the size of the data structures grows roughly linearly in p.

Notable efforts at analysis of the scalability of dense matrix computations include those of Li and Coleman [54] for dense triangular systems, and Saad and Schultz [80]; Ostrouchov, *et al.* [69], and George, Liu, and Ng [37] have made some analyses for algorithms that map matrix columns to processors. Rothberg and Gupta [76] is a important paper for its analysis of the effect of caches on sparse matrix algorithms.

Consider any distributed-memory computation. In order to assess the communication costs analytically, it s useful to employ certain abstract lower bounds. Our model assumes that machine topology is given. It assumes that memory consists of the memories local to processors. It assumes that the communication channels are the edges of a given undirected graph $G = (W, L)$, and that processor–memory units are situated at some, possibly all, of the vertices of the graph. The model includes hypercube and grid-structured message-passing machines, shared-memory machines having physically distributed memory (the Tera machine) as well as tree-structured machines like a CM-5.

Let $V \subseteq W$ be the set of all processors and L be the set of all communication links.

We assume identical links. Let β be the inverse bandwidth (slowness) of a link in seconds per word. (We ignore latency in this model; most large distributed memory computations are bandwidth limited.)

We assume that processors are identical. Let ϕ be the inverse computation rate of a processor in seconds per floating-point operation. Let β_0 be the rate at which a processor can send or receive data, in seconds per word. We expect that β_0 and β will be roughly the same.

A distributed-memory computation consists of a set of processes that exchange information by sending and receiving messages. Let M be the set of all messages communicated. For $m \in M$, $|m|$ denotes the number of words in m. Each message m has a source processor $src(m)$ and a destination processor $dest(m)$, both elements of V.

For $m \in M$, let $d(m)$ denote the length of the path taken by m from the source of the message m to its destination. We assume that each message takes a certain path of links from its source to its destination processor. Let $p(m) = (\ell_1, \ell_2, \ldots, \ell_{d(m)})$ be the path taken by message m. For any link $\ell \in L$, let the set of messages whose paths utilize ℓ, $\{m \in M \mid \ell \in p(m)\}$, be denoted $M(\ell)$.

The following are obviously lower bounds on the completion time of the computation. The first three bounds are computable from the set of message M, each of which is characterized by its size and its endpoints. The last depends on knowledge of the paths $p(M)$ taken by the messages.

1. (Average flux)

$$\frac{{}_{m\in M}|m|\cdot d(m)}{|L|}\cdot\beta.$$

   This is the total flux of data, measured in word-hops, divided by the machine's total communication bandwidth, L/β.

2. (Bisection width) Given $V_0, V_1 \subseteq W$, V_0 and V_1 disjoint, define

$$sep(V_0, V_1) \equiv \min |\{L' \subseteq L \mid L' \text{ is an edge separator of } V_0 \text{ and } V_1\}|$$

   and

$$flux(V_0, V_1) \equiv_{\{m\in M \mid src(m)\in V_i, dest(m)\in V_{1-i}\}} |m|\,.$$

   The bound is

$$\frac{flux(V_0, V_1)}{sep(V_0, V_1)}\cdot\beta.$$

   This is the number of words that cross from one part of the machine to the other, divided by the bandwidth of the wires that link them.

3. (Arrivals/Departures (also known as node congestion))

$$\max_{v\in V}\ dest(m) = v^{|m|\beta_0};$$

$$\max_{v\in V}\ src(m) = v^{|m|\beta_0}.$$

   This is a lower bound on the communication time for the processor with the most traffic into or out of it.

4. (Edge contention)

$$\max_{\ell\in L}{}_{m\in M(\ell)}|m|\beta.$$

   This is a lower bound on the time needed by the most heavily used wire to handle all its traffic.

Of course, the actual communication time may be greater than any of the bounds. In particular, the communication resources (the wires in the machine) need to be scheduled. This can be done dynamically or, when the set of messages is known in advance, statically. With detailed knowledge of the schedule of use of the wires, better bounds can be obtained. For the purposes of analysis of algorithms and assignment of tasks to processors, however, we have found this more realistic approach to be unnecessarily cumbersome. We prefer to use the four bounds above, which depend only on the integrated (i.e. time-independent) information M and, in the case of the edge-contention bound, the paths $p(M)$. In fact, in the work below, we won't assume knowledge of paths and we won't use the edge contention bound.

15.8.2 Cholesky factorization

We'll use the techniques we've introduced to analyze alternative distributed memory implementations of a very important computation, Cholesky factorization of a symmetric, positive definite (SPD) matrix A. The factorization is $A = LL^T$ where L is lower triangular; A is given, L is to be computed.

The algorithm is this:

1. $L := A$
2. `for` $k = 1$ `to` N `do`
3. $L_{kk} := \sqrt{L_{kk}}$
4.   `for` $i = k + 1$ `to` N `do`
5. $L_{ik} := L_{ik} L_{kk}^{-1}$
6.   `for` $j = k + 1$ `to` N `do`
7.   `for` $i = j$ `to` N `do`
8. $L_{ij} := L_{ij} - L_{ik} L_{jk}^T$

We can let the elements L_{ij} be scalars, in which case this is the usual or "point" Cholesky algorithm. Or we can take L_{ij} to be a block, obtained by dividing the rows into contiguous subsets and making the same decomposition of the columns, so that diagonal blocks are square. In the block case, the computation of $\sqrt{L_{kk}}$ (Step 3) returns the (point) Cholesky factor of the SPD block L_{kk}. If A is sparse (has mostly zero entries) then L will be sparse too, although less so than A. In that case, only the non-zero entries in the sparse factor L are stored, and the multiplication/division in lines 5 and 8 are omitted if they compute zeros.

Mapping columns

Assume that the columns of a dense symmetric matrix of order N are mapped to processors cyclically: column j is stored in processor $map(j) \equiv j \bmod p$. Consider communication costs on two-dimensional grid or toroidal machines. Suppose that p is a perfect square and that the machine is a $\sqrt{p} \times \sqrt{p}$ grid. Consider a mapping of the computation in which the operations in line 8 are performed by processor $map(j)$. After performing the operations in line 5, processor $map(k)$ must send column k to all processors $\{map(j) \mid j > k\}$.

Let us fix our attention on 2D grids. There are $L = 2p + O(1)$ links. A column can be broadcast from its source to all other processors through a spanning tree of the machine, a tree of total length p reaching all the processors. Every matrix element will therefore travel over $p - 1$ links, so the total information flux is $(1/2)N^2 p$ and the average flux bound is $(1/4)N^2\beta$.

Only $O(N^2/p)$ words leave any processor. If $N \gg p$, processors must accept almost the whole $(1/2)N^2$ words of L as arriving columns. The bandwidth per processor is β_0, so the arrivals bound is $(1/2)N^2\beta_0$ seconds. If $N \approx p$ the bound drops to half that, $(1/4)N^2\beta_0$ seconds. We summarize these bounds for 2D grids in Table 15.1.

We can immediately conclude that this is a nonscalable distributed algorithm. We may not take $p > \frac{N\phi}{\beta}$ and still achieve high efficiency.

Mapping blocks

Dongarra, Van de Geijn, and Walker [25] have shown that on the Intel Touchstone Delta machine ($p = 528$), mapping blocks is better than mapping columns in LU factorization. In such a mapping, we view the machine as an $p_r \times p_c$ grid and we map elements A_{ij} and L_{ij} to processor

Type of Bound	Lower bound
Arrivals	$\frac{1}{4}N^2\beta_0$
Average flux	$\frac{1}{4}N^2\beta$

Table 15.1: Communication Costs for Column-Mapped Full Cholesky.

Type of Bound	Lower bound
Arrivals	$\frac{1}{4}N^2\beta\left(\frac{1}{p_r}+\frac{1}{p_c}\right)$
Edge contention	$N^2\beta\left(\frac{1}{p_r}+\frac{1}{p_c}\right)$

Table 15.2: Communication Costs for Torus-Mapped Full Cholesky.

$(mapr(i), mapc(j))$. We assume a cyclic mappings here: $mapr(i) \equiv i \bmod p_r$ and similarly for $mapc$.

The analysis of the preceding section may now be done for this mapping. Results are summarized in Table 15.2. With p_r and p_c both $O(\sqrt{p})$, the communication time drops like $O(p^{-1/2})$. With this mapping, the algorithm is scalable even when $\beta \gg \phi$. Now, with $p = O(N^2)$, both the compute time and the communication lower bounds agree; they are $O(N)$. Therefore, we remain efficient when storage per processor is $O(1)$. (This scalable algorithm for distributed Cholesky is due to O'Leary and Stewart [68].)

15.8.3 Distributed sparse Cholesky and the model problem

In the sparse case, the same holds true. To see why this must be true, we need only observe that most of the work in sparse Cholesky factorization takes the form of the factorization of dense submatrices that form during the Cholesky algorithm. Rothberg and Gupta demonstrated this fact in their work in 1992 – 1994.

Unfortunately, with naive cyclic mappings, block-oriented approaches suffer from poor balance of the computational load and modest efficiency. Heuristic remapping of the block rows and columns can remove load imbalance as a cause of inefficiency.

Several researchers have obtained excellent performance using a block-oriented approach, both on fine-grained, massively-parallel SIMD machines [22] and on coarse-grained, highly-parallel MIMD machines [77]. A block mapping maps rectangular blocks of the sparse matrix to processors. A 2-D mapping views the machine as a 2-D $p_r \times p_c$ processor grid, whose members are denoted $p(i, j)$. To date, the 2-D cyclic (also called torus-wrap) mapping has been used: block L_{ij} resides at processor $p(i \bmod p_r, j \bmod p_c)$. All blocks in a given block row are mapped to the same row of processors, and all elements of a block column to a single processor column. Communication volumes grow as the square root of the number of processors, versus linearly for the 1-D mapping; 2-D mappings also asymptotically reduce the critical path length. These advantages accrue even when the underlying machine has some interconnection network whose topology is not a grid.

A 2-D cyclic mapping, however, produces significant load imbalance that severely limits achieved efficiency. On systems (such as the Intel Paragon) with high interprocessor communication bandwidth this load imbalance limits efficiency to a greater degree than communication or want of parallelism.

An alternative, heuristic 2-D block mapping succeeds in reducing load imbalance to a point where it is no longer the most serious bottleneck in the computation. On the Intel Paragon the block mapping heuristic produces a roughly 20% increase in performance compared with the cyclic mapping.

In addition, a scheduling strategy for determining the order in which available tasks are performed adds another 10% improvement.

15.8.4 Parallel Block-Oriented Sparse Cholesky Factorization

In the block factorization approach considered here, matrix blocks are formed by dividing the columns of the $n \times n$ matrix into N contiguous subsets, $N \leq n$. The identical partitioning is performed on the rows. A block L_{ij} in the sparse matrix is formed from the elements that fall simultaneously in row subset i and column subset j.

Each block L_{ij} has an owning processor. The owner of L_{ij} performs all block operations that update L_{ij} (this is the "owner-computes" rule for assigning work). Interprocessor communication is required whenever a block on one processor updates a block on another processor.

Assume that the processors can be arranged as a grid of p_r rows and p_c columns. In a *Cartesian product* (CP) mapping, $map(i,j) = p(RowMap(i), ColMap(j))$, where $RowMap : \{0..N-1\} \rightarrow \{0..p_r-1\}$, and $ColMap : \{0..N-1\} \rightarrow \{0..p_c-1\}$ are given mappings of rows and columns to processor rows and columns. We say that map is *symmetric Cartesian* (SC) if $p_r = p_c$ and $RowMap = ColMap$. The usual 2-D cyclic mapping is SC.[2]

15.9 Load balance with cyclic mapping

Any CP mapping is effective at reducing communication. While the 2-D cyclic mapping is CP, unfortunately it is not very effective at balancing computational load. Experiment and analysis show that the cyclic mapping produces particularly poor load balance; moreover, some serious load balance difficulties must occur for any SC mapping. Improvements obtained by the use of nonsymmetric CP mappings are discussed in the following section.

Our experiments employ a set of test matrices including two dense matrices (DENSE1024 and DENSE2048), two 2-D grid problems (GRID150 and GRID300), two 3-D grid problems (CUBE30 and CUBE35), and 4 irregular sparse matrices from the Harwell-Boeing sparse matrix test set [26]. Nested dissection or minimum degree orderings are used. In all our experiments, we choose $p_r = p_c = \sqrt{P}$, and we use a block size of 48. All Mflops measurements presented here are computed by dividing the operation counts of the best known sequential algorithm by parallel runtimes. Our experiments were performed on an Intel Paragon, using hand-optimized versions of the Level-3 BLAS for almost all arithmetic.

15.9.1 Empirical Load Balance Results

We now report on the efficiency and load balance of the method. Parallel efficiency is given by $t_{seq}/(P \cdot t_{par})$, where t_{par} is the parallel runtime, P is the number of processors, and t_{seq} is the

[2] See [77] for a discussion of *domains*, portions of the matrix mapped in a 1-D manner to further reduce communication.

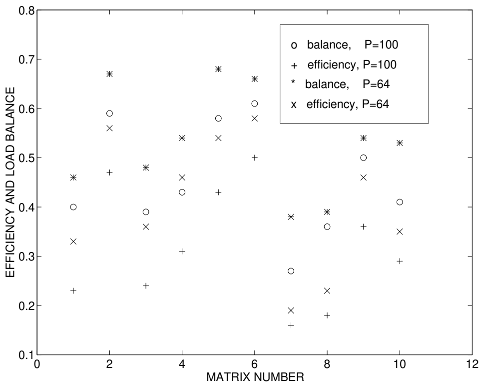


Figure 15.4: Efficiency and overall balance on the Paragon system ($B = 48$).

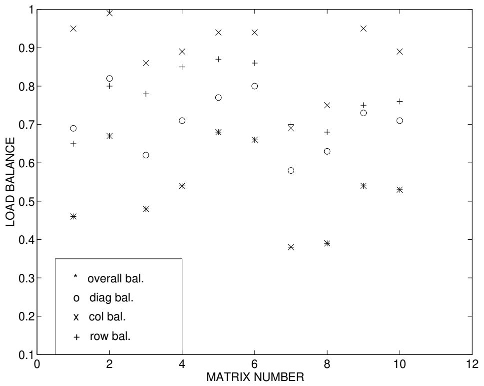


Figure 15.5: Efficiency bounds for 2-D cyclic mapping due to row, column and diagonal imbalances ($P = 64, B = 48$).

runtime for the same problem on one processor. For the data we report here, we measured t_{seq} by factoring the benchmark matrices using our parallel algorithm on one processor. The overall balance of a distributed computation is given by $work_{total}/(P \cdot work_{max})$, where $work_{total}$ is the total amount of work performed in the factorization, P is the number of processors, and $work_{max}$ is the maximum amount of work assigned to any processor. Clearly, overall balance is an upper bound on efficiency.

Figure 1 shows efficiency and overall balance with the cyclic mapping. Observe that load balance and efficiency are generally quite low, and that they are well correlated. Clearly, load balance alone is not a perfect predictor of efficiency. Other factors limit performance. Examples include interprocessor communication costs, which we measured at 5% — 20% of total runtime, long critical paths, which can limit the number of block operations that can be performed concurrently, and poor scheduling, which can cause processors to wait for block operations on other processors to complete. Despite these disparities, the data indicate that load imbalance is an important contributor to reduced efficiency.

We next measured load imbalance among rows of processors, columns of processors, and diagonals of processors. Define $work[i, j]$ to be the runtime due to updating of block L_{ij} by its owner. To approximate runtime, we use an empirically calibrated estimate of the form work = operations + $\omega \cdot$ block-operations; on the Paragon, $\omega = 1,000$.

Define $RowWork[i]$ to be the aggregate work required by blocks in row i: $RowWork[i] =_{j=0}^{N-1} work[i,j]$. An analogous definition applies for $ColWork$, the aggregate column work. Define *row balance* by $work_{total}/p_r \cdot work_{rowmax}$, where $work_{rowmax} = \max_{r} {}_{i:RowMap[i]=r} RowWork[i]$. This row balance statistic gives the best possible overall balance (and hence efficiency), obtained only if there is perfect load balance within each processor row. It isolates load imbalance due to an overloaded processor row caused by a poor row mapping. An analogous expression gives column balance, and a third analogous expression gives diagonal balance. (Diagonal d is made up of the set of processors $p(i,j)$ for which $(i-j) \bmod p_r = d$.) While these three aggregate measures of load balance are only upper bounds on overall balance, the data we present later make it clear that improving these three measures of balance will in general improve the overall load balance.

Figure 2 shows the row, column, and diagonal balances with a 2-D cyclic mapping of the benchmark matrices on 64 processors. Diagonal imbalance is the most severe, followed by row imbalance, followed by column imbalance.

These data can be better understood by considering dense matrices as examples (although the following observations apply to a considerable degree to sparse matrices as well). Row imbalance is due mainly to the fact that $RowWork[i]$, the amount of work associated with a row of blocks, increases with increasing i. More precisely, since $work[i,j]$ increases linearly with j and the number of blocks in a row increases linearly with i, it follows that $RowWork[i]$ increases quadratically in i. Thus, the processor row that receives the last block row in the matrix receives significantly more work than the processor row immediately following it in the cyclic ordering, resulting in significant row imbalance. Column imbalance is not nearly as severe as row imbalance. The reason, we believe, is that while the work associated with blocks in a column increases linearly with the column number j, the number of blocks in the column *decreases* linearly with j. As a result, $ColWork[j]$ is neither strictly increasing nor strictly decreasing. In the experiments, row balance is indeed poorer than column balance. Note that the reason for the row and column imbalance is not that the 2-D cyclic mapping is an SC mapping; rather, we have significant imbalance because the mapping functions $RowMap$ and $ColMap$ are each poorly chosen.

To better understand diagonal imbalance, one should note that blocks on the diagonal of the matrix are mapped exclusively to processors on the main diagonal of the processor grid. Blocks just below the diagonal are mapped exclusively to processors just below the main diagonal of the processor grid. These diagonal and sub-diagonal blocks are among the most work-intensive blocks in the matrix. In sparse problems, moreover, the diagonal blocks are the only ones that are guaranteed to be dense. (For the two dense test matrices, diagonal balance is not significantly worse than row balance.) The remarks we make about diagonal blocks and diagonal processors apply to *any* SC mapping, and do not depend on the use of a cyclic function $RowMap(i) = i \bmod p_r$.

15.10 Heuristic Remapping

Nonsymmetric CP mappings, which map rows independently of columns, are a way to avoid diagonal imbalance that is automatic with SC mappings. We shall choose the row mapping $RowMap$ to maximize the row balance, and independently choose $ColMap$ to maximize column balance. Since the row mapping has no effect on the column balance, and vice versa, we may choose the row mapping in order to maximize row balance independent of the choice of column mapping.

The problems of determining $RowMap$ and $ColMap$ are each cases of a standard NP-complete problem, *number partitioning* [34], for which a simple heuristic is known to be good [3]. This

[3]Number partitioning is a well studied NP-complete problem. The objective is to distribute a set of numbers

heuristic obtains a row mapping by considering the block rows in some predefined sequence. For each processor row, it maintains the total work for all blocks mapped to that row. The algorithm iterates over block rows, mapping a block row to the processor row that has received the least work thus far. We have experimented with several different sequences, the two best of which we now describe.

The **Decreasing Work (DW)** heuristic considers rows in order of decreasing work. This is a standard approach to number partitioning; that small values toward the end of the sequence allow the algorithm to lessen any imbalance caused by large values encountered early in the sequence.

The **Increasing Depth (ID)** heuristic considers rows in order of increasing depth in the elimination tree. In a sparse problem, the work associated with a row is closely related to its depth in the elimination tree.

The effect of these schemes is dramatic. If we look at the three aggregate measures of load balance, we find that these heuristics produce row and column balance of 0.98 or better, and diagonal balance of 0.93 or better, for test case BCSSTK31, which is typical. With ID as the row mapping we have produced better than a 50% improvement in overall balance, and better than a 20% improvement in performance, on average over our test matrices, with $P = 100$. The DW heuristic produces only slightly less impressive improvements The choice of column mapping, as expected, is less important. In fact, for our test suite, the cyclic column mapping and ID row mapping gave the best mean performance. [4]

We also applied these ideas to four larger problems: DENSE4096, CUBE40, COPTER2 (a helicopter rotor blade model, from NASA) and 10FLEET (a linear programming formulation of an airline fleet assignment problem, from Delta Airlines). On 144 and 196 processors the heuristic (increasing depth on rows and cyclic on columns) again produces a roughly 20% performance improvement over a cyclic mapping. Peak performance of 2.3 Gflops for COPTER2 and 2.7 Gflops for 10FLEET were achieved; for the model problems CUBE40 and DENSE4096 the speeds were 3.2 and 5.2 Gflops.

In addition to the heuristics described so far, we also experimented with two other approaches to improving factorization load balance. The first is a subtle modification of the original heuristic. It begins by choosing some column mapping (we use a cyclic mapping). This approach then iterates over rows of blocks, mapping a row of blocks to a row of processors so as to minimize the amount of work assigned to any one *processor*. Recall that the earlier heuristic attempted to minimize the aggregate work assigned to an entire row of processors. We found that this alternative heuristic produced further large improvements in overall balance (typically 10-15% better than that of our original heuristic). Unfortunately, realized performance did not improve. This result indicates that load balance is not the most important performance bottleneck once our original heuristic is applied.

A very simple alternate approach reduces imbalance by performing cyclic row and column mappings on a processor grid whose dimensions p_c and p_r are relatively prime; this reduces diagonal imbalance. We tried this using 7×9 and 9×11 processor grids (using one fewer processor that for our earlier experiments with $P = 64$ and $P = 100$.) The improvement in performance is somewhat lower than that achieved with our earlier remapping heuristic (17% and 18% mean improvement on 63 and 99 processors versus 20% and 24% on 64 and 100 processors). On the other hand, the mapping needn't be computed.

---

among a fixed number of bins so that the maximum sum in any bin is minimized.

[4]Full experimental data has appeared in another paper [78].

15.11 Scheduling Local Computations

The next questions to be addressed, clearly, are: (i) what is the most constraining bottleneck after our heuristic is applied, and (ii) can this bottleneck be addressed to further improve performance?

One potential remaining bottleneck is communication. Instrumentation of our block factorization code reveals that on the Paragon system, communication costs account for less than 20% of total runtime for all problems, even on 196 processors. The same instrumentation reveals that most of the processor time not spent performing useful factorization work is spent idle, waiting for the arrival of data.

We do not believe that the idle time is due to insufficient parallelism. Critical path analysis for problem BCSSTK15 on 100 processors, for example, indicates that it should be possible to obtain nearly 50% higher performance than we are currently obtaining. The same analysis for problem BCSSTK31 on 100 processors indicates that it should be possible to obtain roughly 30% higher performance. We therefore suspected that the scheduling of tasks by our code was not optimal.

To that end we tried alternative scheduling policies. They are:

FIFO. Tasks are initiated in the order in which the processor discovers that they are ready.

Destination depth. Ready tasks initiated in order of the destination block's elimination tree depth.

Source depth. Ready tasks initiated in order of the source block's elimination tree depth.

For the FIFO policy, a queue of ready tasks is used, while for the others, a heap is used. We experimented with 64 processors, using BSCCST31, BCSSTK33, and DENSE2048. Both priority-based schemes are better than FIFO; destination depth seems slightly better than source depth. We observed a slowdown of 2% due to the heap data structure on BCSSTK33; the destination priority scheme then improved performance by 15% for a net gain of 13%. For BSCCST31 the net gain was 8%. For DENSE2048, however, there was no gain. This improvement is encouraging. There may be more that can be achieved through the pursuit of a better understanding of the scheduling question.

Chapter 16

Domain Decomposition for PDE

Domain decomposition is a term used by at least two different communities. Literally, the words indicate the partitioning of a region. As we will see in Part ?? of this book, an important computational geometry problem is to find good ways to partition a region. This is not what we will discuss here.

In scientific computing, domain decomposition refers to the technique of solving partial differential equations using subroutines that solve problems on subdomains. Originally, a domain was a contiguous region in space, but the idea has generalized to include any useful subset of the discretization points. Because of this generalization, the distinction between domain decomposition and multigrid has become increasingly blurred.

Domain decomposition is an idea that is already useful on serial computers, but it takes on a greater importance when one considers a parallel machine with, say, a handful of very powerful processors. In this context, domain decomposition is a parallel divide-and-conquer approach to solving the PDE.

To guide the reader, we quickly summarize the choice space that arises in the domain decomposition literature. As usual a domain decomposition problem starts as a continuous problem on a region and is disretized into a finite problem on a discrete domain.

We will take as our model problem the solution of the elliptic equation $\nabla^2 u = f$ on a region Ω which is the union of at least subdomains Ω_1 and Ω_2. Domain decomposition ideas tend to be best developed for elliptic problems, but may be applied in more general settings.

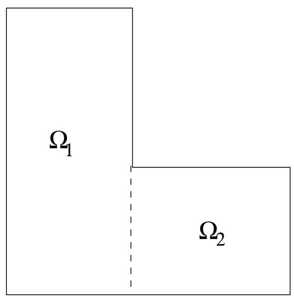


Figure 16.1: Domain divided into two subdomains without overlap

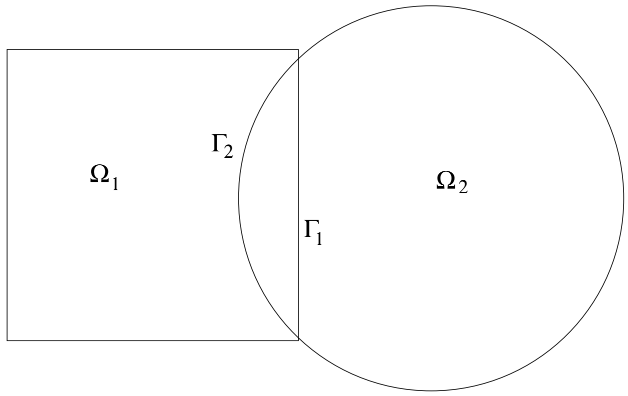


Figure 16.2: Example domain of circle and square with overlap

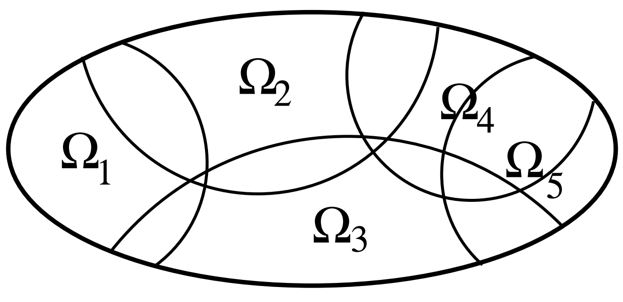


Figure 16.3: Example domain with many overlapping regions

DOMAIN DECOMPOSITION OUTLINE

1. Geometric Issues
   - Overlapping or non-overlapping regions
   - Geometric Discretization
     - Finite Difference or Finite Element
     - Matching or non-matching grids
2. Algorithmic Issues
   - Algebraic Discretization
     - Schwarz Approaches: Additive vs Multiplicative
     - Substructuring Approaches
   - Accelerants
     - Domain Decomposition as a Preconditioner
     - Course (Hierarchical/Multilevel) Domains
3. Theoretical Considerations

16.1 Geometric Issues

The geometric issues in domain decomposition are 1) how are the domains decomposed into subregions, and 2) how is the region discretized using some form of grid or irregular mesh. We consider these issues in turn.

16.1.1 Overlapping vs Non-overlapping regions

So as to emphasize the issue of overlap vs non-overlap, we can simplify all the other issues by assuming that we are solving the continuous problem (no discretization) exactly on each domain (no choice of algorithm). The reader may be surprised to learn that domain decomposition methods divide neatly into either being overlapping or nonoverlapping methods. Though one can find much in common between these two methods, they are really rather different in flavor. When the methods overlap, the methods are sometimes known as Schwarz methods, while when there is no overlap, the methods are sometimes known as substructuring. (Historically, the former name was used in the continuous case, and the latter in the discrete case, but this historical distinction has been. and even should be, blurred.)

We begin with the overlapping region illustrated in Figure 16.2. Schwarz in 1870 devised an obvious alternating procedure for solving Poisson's equation $\nabla^2 u = f$:

1. Start with any guess for u_2 on Γ_1.
2. Solve $\nabla^2 u = f$ on Ω_1 by taking $u = u_2$ on Γ_1. (i.e. solve in the square using boundary data from the interior of the circle)
3. Solve $\nabla^2 u = f$ on Ω_2 by taking $u = u_1$ on Γ_2 (i.e. solve in the circle using boundary data from the interior of the square)
4. Goto 2 and repeat until convergence is reached

One of the characteristics of elliptic PDE's is that the solution at every point depends on global conditions. The information transfer between the regions clearly occurs in the overlap region.

If we "choke" the transfer of information by considering the limit as the overlap area tends to 0, we find ourselves in a situation typified by Figure 16.1. The basic Schwarz procedure no longer works? Do you see why? No matter what the choice of data on the interface, it would not be updated. The result would be that the solution would not be differentiable along the interface.

One approach to solving the non-overlapped problem is to concentrate directly on the domain of intersection. Let g be a current guess for the solution on the interface. We can then solve $\nabla^2 u = f$ on Ω_1 and Ω_2 independently using the value of g as Dirichlet conditions on the interface. We can define the map

$$T : g \to \frac{\partial g}{\partial n_1} + \frac{\partial g}{\partial n_2}.$$

This is a map from functions on the interface to functions on the interface defined by taking a function to the jump in the derivative. The operator T is known as the Steklov-Poincaré operator.

Suppose we can find the exact solution to $Tg = 0$. We would then have successfully decoupled the problem so that it may be solved independly into the two domains Ω_1 and Ω_2. This is a "textbook" illustration of the divide and conquer method, in that solving $Tg = 0$ constitutes the "divide."

16.1.2 Geometric Discretization

In the previous section we contented ourselves with formulating the problem on a continuous domain, and asserted the existence of solutions either to the subdomain problems in the Schwarz case, or the Stekhlov-Poincaré operator in the continuous case.

Of course on a real computer, a discretizatization of the domain and a corresponding discretization of the equation is needed. The result is a linear system of equations.

Finite Differences or Finite Elements

Finite differences is actually a special case of finite elements, and all the ideas in domain decomposition work in the most general context of finite elements. In finite differences, one typically imagines a square mesh. The prototypical example is the five point stencil for the Laplacian in two dimensions. An analog computer to solve this problem would consist of a grid of one ohm resistors. In finite elements, the protypical example is a triangulation of the region, and the appropriate formulation of the PDE on these elements.

Matching vs Non-matching grids

When solving problems as in our square-circle example of Figure 16.2, it is necessary to discretize the interior of the regions with either a finite difference style grid or a finite element style mesh. The square may be nicely discretized by covering it with Cartesian graph-paper, while the circle may be more conveniently discretized by covering it with Polar graph-paper. Under such a situation, the grids do not match, and it becomes necessary to transfer points interior to Ω_2 to the boundary of Ω_1 and vice versa.

16.2 Algorithmic Issues

Once the domain is discretized, numerical algorithms must be formulated. There is a definite line drawn between Schwarz (overlapping) and substructuring (non-overlapping) approaches.

16.2.1 Schwarz approaches: additive vs multiplicative

A procedure that alternates between solving an equation in Ω_1 and then Ω_2 does not seem to be parallel at the highest level because if processor 1 contains all of Ω_1 and processor 2 contains all of Ω_2 then each processor must wait for the solution of the other processor before it can execute. Such approaches are known as multiplicative approaches because of the form of the operator applied to the error. Alternatively, approaches that allow for the solution of subproblems simultaneously are known as additive methods. The difference is akin to the difference between Jacobi and Gauss-Seidel. We proceed to explain this in further detail.

Classical Iterations and their block equivalents

Let us review the basic classical methods for solving pde's on a discrete domain.

1. Jacobi - At step n, the neighboring values used are from step $n-1$
2. Gauss-Seidel - Values at step n are used if available, otherwise the values are used from step $n-1$
3. Red Black Ordering - If the grid is a checkerboard, solve all red points in parallel using black values at $n-1$, then solve all black points in parallel using red values at step n

Analogous *block* methods may be used on a domain that is decomposed into a number of multiple regions. Each region is thought of as an element used to solve the larger problem. This is known as block Jacobi, or block Gauss-Seidel.

1. Gauss-Seidel - Solve each region in series using the boundary values at n if available.
2. Jacobi - Solve each region on a separate processor in parallel and use boundary values at $n-1$. (Additive scheme)
3. Coloring scheme - Color the regions so that like colors do not touch and solve all regions with the same color in parallel. (Multiplicative scheme)

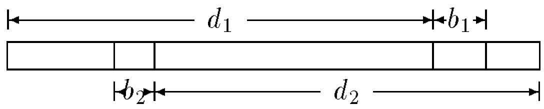
The block Gauss-Seidel algorithm is called a multiplicative scheme for reasons to be explained shortly. In a corresponding manner, the block Jacobi scheme is called an additive scheme.

Overlapping regions: A notational nightmare?

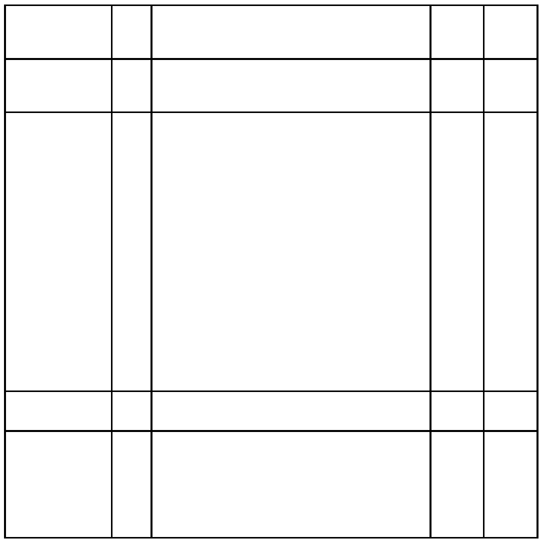
When the grids match it is somewhat more convenient to express the discretized PDE as a simple matrix equation on the gridpoints.

Unfortunately, we have a notational difficulty at this point. It is this difficulty that is probably the single most important reason that domain decomposition techniques are not used as extensively as they can be. Even in the two domain case, the difficulty is related to the fact that we have domains 1 and 2 that overlap each other and have internal and external boundaries. By setting the boundary to 0 we can eliminate any worry of external boundaries. I believe there is only one reasonable way to keep the notation manageable. We will use subscripts to denote subsets of indices. d_1 and d_2 will represent those nodes in domain 1 and domain 2 respectively. b_1 and b_2 will represent those notes in the boundary of 1 and 2 respectively that are not external to the entire domain.

Therefore u_{d_1} denotes the subvector of u consisting of those elements interior to domain 1, while A_{u_1,b_1} is the rectangular subarray of A that map the interior of domain 1 to the internal boundary of domain 1. If we were to write u^T as a row vector, the components might break up as follows (the overlap region is unusually large for emphasis:)



Correspondingly, the matrix A (which of course would never be written down) has the form



The reader should find A_{b_1,b_1} etc., on this picture. To further simplify notation, we write 1 and 2 for d_1 and d_2,1_b and 2_b for b_1 and b_2, and also use only a single index for a diagonal block of a matrix (i.e. $A_1 = A_{11}$).

Now that we have leisurely explained our notation, we may return to the algebra. Numerical analysts like to turn problems that may seem new into ones that they are already familiar with. By carefully writing down the equations for the procedure that we have described so far, it is possible to relate the classical domain decomposition method to an iteration known as *Richardson iteration.* Richardson iteration solves $Au = f$ by computing $u^{k+1} = u^k + M(f - Au^k)$, where M is a "good" approximation to A^{-1}. (Notice that if $M = A^{-1}$, the iteration converges in one step.)

The iteration that we described before may be written algebraically as

$$A_1 u_1^{k+1/2} + A_{1,1_b} u_{1_b}^k = f_1$$

$$A_2 u_2^{k+1} + A_{2,2_b} u_{2_b}^{k+1/2} = f_2$$

Notice that values of $u^{k+1/2}$ updated by the first equation, specifically the values on the boundary of the second region, are used in the second equation.

With a few algebraic manipulations, we have

$$u_1^{k+1/2} = u_1^{k-1} + A_1^{-1}(f - Au^{k-1})_1$$

$$u_2^{k+1} = u_2^{k+1/2} + A_2^{-1}(f - Au^{k+1/2})_2$$

This was already obviously a Gauss-Seidel like procedure, but those of you familiar with the algebraic form of Gauss-Seidel might be relieved to see the form here.

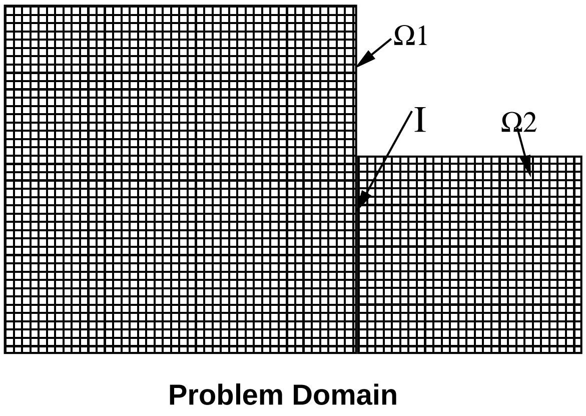


Figure 16.4: Problem Domain

A roughly equivalent block Jacobi method has the form

$$u_1^{k+1/2} = u_1^{k-1} + A_1^{-1}(f - Au^{k-1})_1$$

$$u_2^k = u_2^{k+1/2} + A_2^{-1}(f - Au^k)_2$$

It is possible to eliminate $u^{k+1/2}$ and obtain

$$u^{k+1} = u^k + (A_1^{-1} + A_2^{-1})(f - Au^k),$$

where the operators are understood to apply to the appropriate part of the vectors. It is here that we see that the procedure we described is a Richardson iteration with operator $M = A_1^{-1} + A_2^{-1}$.

16.2.2 Substructuring Approaches

Figure 16.4 shows an example domain of a problem for a network of resistors or a discretized region in which we wish to solve the Poisson equation, We will see that the discrete version of the Steklov-Poincaré operator has its algebraic equivalent in the form of the Schur complement.

$\nabla^2 v = g$

In matrix notation, $Av = g$, where

$$A = \begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & A_2 & A_{2I} \\ A_{I1} & A_{I2} & A_I \end{pmatrix}$$

One of the direct methods to solve the above equation is to use LU or LDU factorization. We will do an analogous procedure with blocks. We can rewrite A as,

$$A = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ A_{I1}A_1^{-1} & A_{I2}A_2^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S \end{pmatrix} \begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & A_2 & A_{2I} \\ 0 & 0 & I \end{pmatrix}$$

where,

$$S = A_I - A_{I1}A_1^{-1}A_{1I} - A_{I2}A_2^{-1}A_{2I}$$

We really want A^{-1}

$$A^{-1} = \begin{pmatrix} A_1^{-1} & 0 & -A_1^{-1}A_{1I} \\ 0 & A_2^{-1} & -A_2^{-1}A_{2I} \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ -A_{I1}A_1^{-1} & -A_{I2}A_2^{-1} & I \end{pmatrix} \tag{16.1}$$

Inverting S turns out to be the hardest part.

$$A^{-1} \begin{pmatrix} V_{\Omega 1} \\ V_{\Omega 2} \\ V_{Interface} \end{pmatrix} \begin{matrix} \rightarrow Voltages & in & region & \Omega 1 \\ \rightarrow Voltages & in & region & \Omega 2 \\ \rightarrow Voltages & at & interface & \end{matrix}$$

Let us examine Equation 16.1 in detail.

In the third matrix,
A_1^{-1} - Poisson solve in $\Omega 1$
A_{I1} - is putting the solution onto the interface
A_2^{-1} - Poisson solve in $\Omega 2$
A_{I2} - is putting the solution onto the interface

In the second matrix,
Nothing happening in domain 1 and 2
Complicated stuff at the interface.

In the first matrix we have,
A_1^{-1} - Poisson solve in $\Omega 1$
A_2^{-1} - Poisson solve in $\Omega 2$
$A_1^{-1}A_{1I}$ and $A_2^{-1}A_{1I}$- Transferring solution to interfaces

In the above example we had a simple 2D region with neat squares but in reality we might have to solve on complicated 3D regions which have to be divided into tetrahedra with 2D regions at the interfaces. The above concepts still hold.

Getting to S^{-1},

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ c/a & 1 \end{pmatrix} \begin{pmatrix} a & b \\ 0 & d - bc/a \end{pmatrix}$$

where, $d - bc/a$ is the Schur complement of d.

In Block form

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ CA^{-1} & 1 \end{pmatrix} \begin{pmatrix} A & B \\ 0 & D - CA^{-1}B \end{pmatrix}$$

We have

$$S = A_I - A_{I1}A_1^{-1}A_{1I} - A_{I2}A_2^{-1}A_{2I}$$

Arbitrarily break A_I as

$$A_I = A_I^1 + A_I^2$$

Think of A as

$$\begin{pmatrix} A_1 & 0 & A_{1I} \\ 0 & 0 & 0 \\ A_{I1} & 0 & A_I^1 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & A_2 & A_{2I} \\ 0 & A_{I2} & A_I^2 \end{pmatrix}$$

Schur Complements are

$$S^1 = A_I^1 - A_{I1}A_1^{-1}A_{1I}$$

$$S^2 = A_I^2 - A_{I2}A_2^{-1}A_{2I}$$

and

$$S = S^1 + S^2$$

$A_1^{-1} \rightarrow$ Poisson solve on $\Omega 1$
$A_2^{-1} \rightarrow$ Poisson solve on $\Omega 2$
A_{I1} $\Omega 1 \rightarrow I$
A_{21} $\Omega 2 \rightarrow I$
A_{1I} $I \rightarrow \Omega 1$
A_{2I} $I \rightarrow \Omega 2$

Sv - Multiplying by the Schur Complement involves 2 Poisson solves and some cheap transferring.

$S^{-1}v$ should be solved using Krylov methods. People have recommended the use of S_1^{-1} or S_2^{-1} or $(S_1^{-1} + S_2^{-1})$ as a preconditioner

16.2.3 Accerlerants

Domain Decomposition as a Preconditioner

It seems wasteful to solve subproblems extremely accurately during the early stages of the algorithm when the boundary data is likely to be fairly inaccurate. Therefore it makes sense to run a few steps of an iterative solver as a preconditioner for the solution to the entire problem.

In a modern approach to the solution of the entire problem, a step or two of block Jacobi would be used as a preconditioner in a Krylov based scheme. It is important at this point not to lose track what operations may take place at each level. To solve the subdomain problems, one might use multigrid, FFT, or preconditioned conjugate gradient, but one may choose to do this approximately during the early iterations. The solution of the subdomain problems itself may serve as a preconditioner to the solution of the global problem which may be solved using some Krylov based scheme.

The modern approach is to use a step of block Jacobi or block Gauss-Seidel as a preconditioner for use in a Krylov space based subsolver. There is not too much point in solving the subproblems exactly on the smaller domains (since the boundary data is wrong) just an approximate solution suffices $\rightarrow$ **domain decomposition preconditioning**

Krylov Methods - Methods to solve linear systems : **A**u=g . Examples have names such as the Conjugate Gradient Method, GMRES (Generalized Minimum Residual), BCG (Bi Conjugate Gradient), QMR (Quasi Minimum Residual), CGS (Conjugate Gradient Squared). For this lecture, one can think of these methods in terms of a black-box. What is needed is a subroutine that given u computes Au. This is a matrix-vector multiply in the abstract sense, but of course it is not a dense matrix-vector product in the sense one practices in undergraduate linear algebra. The other needed ingredient is a subroutine to *approximately* solve the system. This is known as a preconditioner. To be useful this subroutine must roughly solve the problem quickly.

Course (Hierarchical/Multilevel) Techniques

These modern approaches are designed to greatly speed convergence by solving the problem on different sized grids with the goal of communicating information between subdomains more efficiently. Here the "domain" is a course grid. Mathematically, it is as easy to consider a contiguous domain consisting of neighboring points, as it is is to consider a course grid covering the whole region.

Up until now, we saw that subdividing a problem did not directly yield the final answer, rather it simplified or allowed us to change our approach in tackling the resulting subproblems with existing methods. It still required that individual subregions be composited at each level of refinement to establish valid conditions at the interface of shared boundaries.

Multilevel approaches solve the problem using a coarse grid over each sub-region, gradually accommodating higher resolution grids as results on shared boundaries become available. Ideally for a well balanced multi-level method, no more work is performed at each level of the hierarchy than is appropriate for the accuracy at hand.

In general a hierarchical or multi-level method is built from an understanding of the difference between the damping of low frequency and high components of the error. Roughly speaking one can kill of low frequency components of the error on the course grid, and higher frequency errors on the fine grid.

Perhaps this is akin to the Fast Multipole Method where p poles that are "well-separated" from a given point could be considered as clusters, and those nearby are evaluated more precisely on a finer grid.

16.3 Theoretical Issues

This section is not yet written. The rough content is the mathematical formulation that identifies subdomains with projection operators.

16.4 A Domain Decomposition Assignment: Decomposing MIT

Perhaps we have given you the impression that entirely new codes must be written for parallel computers, and furthermore that parallel algorithms only work well on regular grids. We now show you that this is not so.

You are about to solve Poisson's equation on our MIT domain:

Notice that the letters MIT have been decomposed into 32 rectangles – this is just the right number for solving

$$\frac{\partial^2 u}{dx^2} + \frac{\partial^2 u}{dy^2} = \rho(x, y)$$

on a 32 processor CM-5.

To solve the Poisson equation on the individual rectangles, we will use a FISHPACK library routine. (I know the French would cringe, but there really is a library called FISHPACK for solving the Poisson equation.) The code is old enough (from the 70's) but in fact it is too often used to really call it dusty.

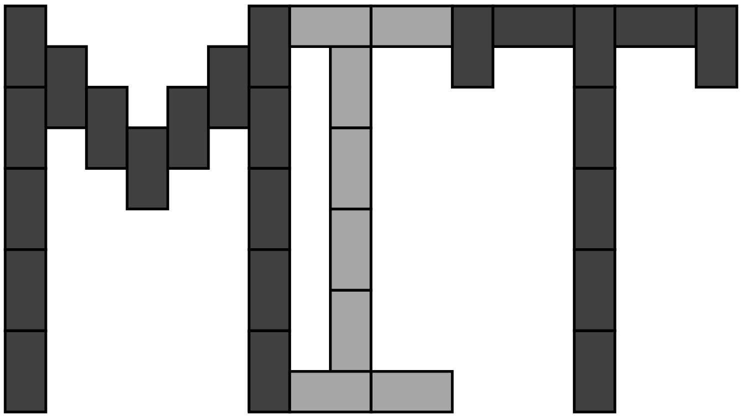


Figure 16.5: MIT domain

[Can you tell that the paragraphs below were written before the web caught on?]

As a side point, this exercise highlights the ease of grabbing kernel routines off the network these days. High quality numerical software is out there (bad stuff too). One good way to find it is via `xnetlib`.

On athena, type

```
athena% add netlib
athena% xnetlib
```

to play with netlib which is actually running off of a machine in Oak Ridge, TN. Click on `Library`, then `fishpack`, then the first routine: `fishpack/hwscrt.f`

Now click `download` on the top bar button, then "Get Files Now" as you watch your gas tank go from empty to full. Xnetlib has now created the directory `xnlFiles` (x-net-lib-files). You can `cd xnlFiles/fishpack` to see the routine.

Congratulations, you have just obtained free software over the network. You probably want to have a Friday directory on scout. You can then ftp the code over to scout by typing on athena:

```
ftp scout.lcs.mit.edu
```

and using your userid (`mit-0?`) followed by your password. (If you hit return after your last name it will not work.) If all the directories are set up right you can type `put hwscrt.f` to move the file to scout.

All of the rectangles on the MIT picture have sides in the ratio 2 to 1; some are horizontal while others are vertical. We have arbitrarily numbered the rectangles accoding to scheme below, you might wish to write the numbers in the picture on the first page.

```
 4                 10 21 21 22 22 23 24 24 25 26 26 27
 4  5           9  10    20       23       25       27
 3  5  6     8  9  11    20                28
 3     6  7  8     11    19                28
 2        7        12    19                29
 2                 12    18                29
 1                 13    18                30
 1                 13    17                30
```

```
    0                     14    17                    31
    0                     14 15 15 16 16              31
```

In our file neighbor.data which you can take from `~edelman/summer94/friday` we have encoded information about neighbors and connections. A copy of the file is included in this handout. You will see numbers such as

```
1  0  0  0  0  0
4  0  0  0  0  0
0  0  0  0
0
```

This contains information about the 0th rectangle. The first line says that it has a neighbor 1. The 4 means that the neighbor meets the rectangle on top. (1 would be the bottom, 6 would be the lower right.) We starred out a few entries towards the bottom. Figure out what they should be.

In the actual code (solver.f), a few lines were question marked out for the message passing. Figure out how the code works and fill in the appropriate lines. The program may be compiled with the makefile.

Chapter 17

Multilevel Methods

Multilevel methods can be viewed a family of useful methods for solving sparse linear systems defined on spatial domains. It can also be viewed as an extension of our basic numerical methods such as the finite element and finite difference methods. Instead using a single mesh to approximate PDEs on a physical domain as in the finite element and difference method, the multilevel method (or called hierarchical methods) use a gradient (series) of meshes M_0, ..., M_k, where M_i is a coarsened mesh of M_{i-1}, or equivalently M_{i-1} is an refinement of M_i . Two examples are multigrid (MD) [?] and (multilevel) domain decomposition (DD) [?]. Both methods are iterative in natural. They perform an iterative computation that transforms partial solutions from one mesh to its coarsened mesh or refined mesh (called its neighboring meshes). It uses interpolation or restriction to go from mesh to mesh.

17.1 Multigrids

Originally, multigrid methods were developed to solve boundary value problems posed on a (geometric) physical domain. Therefore, there are two important inputs: the PDEs and the geometric structure of the domain. Such problems are made discrete by choosing a set of grid points.

17.1.1 The Basic Idea

1. Use coarse grids to obtain better initial guesses for the finest grid.
2. Use coarse grids to correct the error of the solution obtained on the finest grid. We call such a scheme a coarse grid correction scheme.

17.1.2 Restriction and Interpolation

For supporting both approach, we need to define

- an operator to transfer a fine grid vector to a coarse grid vector.
- an operator to a coarse grid vector to a fine grid vector.
- the corresponding linear systems for coarse grids.

To move the solution from a coarse grid to a fine one, we use *interpolation.* Conversely, to move the solution from a fine grid to a coarse one we apply *restriction.* Many methods for interpolation and restriction can be used. Here we discuss some simple ones that are commonly used in multigrid implementation. *These simple ones are quite effective in practice.*

Let I_{2h}^{h} denote the interpolation operator. It takes a coarse grid vector v^{2h} and return a fine grid vector v^{h}, i.e., $I_{2h}^{h}v^{2h} = v^{h}$. For one dimensional fine grid of $n-1$ points , the following interpolation vector is the most commonly used one:

$$\begin{aligned} v_{2j}^{h} &= v_{j}^{2h} \\ v_{2j+1}^{h} &= (v_{j}^{2h} + v_{j+1}^{2h})/2, \end{aligned}$$

where $0 \leq j \leq n/2-1$.

Notice that the interpolation operator is an linear operator and can be written as a matrix vector product. We else use I_{2h}^{h} to denote the transformation matrix.

For two dimensional grids, the interpolation operator can be defined in a similar way. One again, we let I_{2h}^{h} denote the interpolation operator and hence, $I_{2h}^{h}v^{2h} = v^{h}$. The components of v^{h} are then given by

$$\begin{aligned} v_{2i,2j}^{h} &= v_{i,j}^{2h} \\ v_{2i+1,2j}^{h} &= (v_{i,j}^{2h} + v_{i+1,j}^{2h})/2 \\ v_{2i,2j+1}^{h} &= (v_{i,j}^{2h} + v_{i,j+1}^{2h})/2 \\ v_{2i+1,2j+1}^{h} &= (v_{i,j}^{2h} + v_{i+1,j}^{2h} + v_{i,j+1}^{2h} + v_{i+1,j+1}^{2h} \end{aligned}$$

where $0 \leq i,j \leq n/2-1$.

Because the regular grids used in multigrid are well-nested. The simplest way to transfer vector from a fine grid to a coarse grid is by injection, i.e., by assign $v_{j}^{2h} = v_{2j}^{h}$ for one dimensional case. An alternative restriction operator is called *full weighting* and we denoted it by I_{h}^{2h}. It takes a fine grid vector v^{h} and return a coarse grid vector v^{2h}, i.e., $I_{h}^{2h}v^{h} = v^{2h}$. The components of v^{h} are given by

$$v_{j}^{2h} = (v_{2j-1}^{h} + 2v_{2j}^{h} + v_{2j+1}^{h})/4,$$

where $1 \leq j \leq n/2-1$.

As a linear operator, we have $I_{2h}^{h} = 2(I_{h}^{2h})^{T}$, where $(I_{h}^{2h})^{T}$ is the transpose of (I_{h}^{2h}).

Similarly, we can define the *full weighting restriction* operator for two dimensional regular grid, $I_{h}^{2h}v^{h} = v^{2h}$. The components of v^{h} are given by

$$\begin{aligned} vj^{2h} = & \{(v_{2i-1,2j-1}^{h} + 2v_{2i-1,2j+1}^{h} + v_{2i+1,2j-1}^{h} + 2v_{2i+1,2j+1}^{h}) \\ & (v_{2i,2j-1}^{h} + 2v_{2i,2j+1}^{h} + v_{2i-1,2j}^{h} + 2v_{2i+1,2j}^{h}) + 4v_{2i,2j}^{h}\} \end{aligned}$$

where $1 \leq i,j \leq n/2-1$. Again, we have $I_{2h}^{h} = 2(I_{h}^{2h})^{T}$. Clearly in theory, we can use other convex combination of the values of the neighboring points in the restriction formulation.

Numerically, the interpolation can restriction operator transfer the vector of errors on one grid to another. So its numerical effectiveness and correctness depends on the assumption that the error is "smooth". Notice that if the error is highly oscillatory, then clearly the interpolation and the full weighted restriction may produce an error vector that is not very accurate. Fortunately, for multigrid, the iterative process at each grid eliminates the oscillatory components of the error vector and hence produce an error vector that is much smooth. The process of interpolation, restriction, and local relaxation complement each other. That is the intuition why multigrid can be very effective.

Clearly, for regular grids, the computation in both interpolation and restriction involves values only from nearest neighbors and can be written again as a sparse matrix time a vector. Therefore such operation can be supported efficiently on parallel machines.

17.1.3 The Multigrid Scheme

To simplify our discussion, we introduction some notation.

- Let A^h denote the linear system associated with grid of spacing h.
- For each vector name, say v, let v^h denote a vector for the grid of spacing h.

Therefore, if h is the spacing of the finest grid, then the linear systems associated with the finest grid is give as $A^h u^h = f^h$.

The simplest multigrid scheme is the use the coarse grid to relax the error. Such a scheme is referred as a V-cycle scheme in the literature.

Algorithm V-cycle MG(v^h, f^h)
if Ω^h is the coarsest grid, solve $A^h u^h = f^h$ directly (if A^h is small enough), or apply an iterative method k times to relax the equation $A^h u^h = f^h$ with initial solution v^h to obtain a new v^h.
else

1. apply an iterative method k_1 times to relax the equation $A^h u^h = f^h$ with initial solution v^h to obtain a new v^h.
2. $r^{2h} = I_h^{2h}(f^h - A^h v^h)$
3. $e^{2h} = 0$
4. $e^{2h} =$ V-cycle MG(e^{2h}, r^{2h})
5. $e^h = I_{2h}^h(e^{2h})$
6. $v^h = v^h + e^h$
7. apply an iterative method k_2 times to relax the equation $A^h u^h = f^h$ with initial solution v^h to obtain a new v^h.

Suppose u^* is the exact solution to approximation e^h of $e.A^h u^h = f^h$. The basic idea here is to first solve the finest grid linear equation $A^h u^h = f^h$ iteratively with an initial guess v^h to obtain a new v^h. Therefore the error of the solution is $e = u^* - v^h$ and the residual error is $r^h = f^h - A^h v^h$. Therefore $u^* = v^h + e$. We also know that $A^h e = fr^h$. Instead to find e on the finest grid, we transfer r^h to $r^2 h$ and try to approximate $A^{2h} e^2 h = r^{2h}$ on the coarse grid (recursively). After obtain an approximation e^{2h}, we transfer e^{2h} back to the finer grid to obtain an approximation of e^h. The new approximation to the linear system is then $v^h + e^h$.

V Graph and Geometric Algorithms

Chapter 18

Partitioning and Load Balancing

Handling a large mesh or a linear system on a supercomputer or on a workstation cluster usually requires that the data for the problem be somehow partitioned and distributed among the processors. The quality of the partition affects the speed of solution: a good partition divides the work up evenly and requires as little communication as possible. Unstructured meshes may approximate irregular physical problems with fewer mesh elements, their use increases the difficulty of programming and requires new algorithms for partitioning and mapping data and computations onto parallel machines. Partitioning is also important for VLSI circuit layout and parallel circuit simulation.

18.1 Motivation from the Parallel Sparse Matrix Vector Multiplication

Multiplying a sparse matrix by a vector is a basic step of most commonly used iterative methods (conjugate gradient, for example). We want to find a way of efficiently parallelizing this operation.

Say that processor i holds the value of v_i. To update this value, processor need to compute a weighted sum of values at itself and all of its neighbors. This means it has to first receiving values from processors holding values of its neighbors and then computing the sum. Viewing this in graph terms, this corresponds to communicating with a node's nearest neighbors.

We therefore need to break up the vector (and implicitly matrix and graph) so that:

- We balance the computational load at each processor. This is directly related to the number of non-zero entries in its matrix block.
- We minimize the communication overhead. How many other values does a processor have to receive? This equals the number of these values that are held at other processors.

We must come up with a proper division to reduce overhead. This corresponds to dividing up the graph of the matrix among the processors so that there are very few crossing edges. First assume that we have 2 processors, and we wish to partition the graph for parallel processing. As an easy example, take a simplistic cut such as cutting the 2D regular grid of size n in half through the middle. Let's define the cut size as the number of edges whose endpoints are in different groups. A good cut is one with a small cut size. In our example, the cut size would be $\sqrt{n}$. Assuming that the cost of each communication is 10 times more than an local arithmetic operation. Then the total parallel cost of perform matrix-vector product on the grid is $(4n)/2 + 10\sqrt{n} = 2n + 10\sqrt{n}$. In general, for p processors, to need to partition the graph into p subgraphs.

Various methods are used to break up a graph into parts such that the number of crossing edges is minimized. Here we'll examine the case where p = 2 processors, and we want to partition the graph into 2 halves.

18.2 Separators

The following definitions will be of use in the discussion that follows. Let $G = (V, E)$ be an undirected graph.

- A **bisection** of G is a division of V into V_1 and V_2 such that $|V_1| = |V_2|$. (If $|V|$ is odd, then the cardinalities differ by at most 1). The **cost** of the bisection (V_1, V_2) is the number of edges connecting V_1 with V_2.
- If $\beta \in [1/2, 1)$, a β-**bisection** is a division of V such that $V_{1,2} \leq \beta|V|$.
- An **edge separator** of G is a set of edges that if removed, would break G into 2 pieces with no edges connecting them.
- A p-**way partition** is a division of V into p pieces V_1,V_2,. . .,V_p where the sizes of the various pieces differ by at most 1. The cost of this partition is the total number of edges crossing the pieces.
- A **vertex separator** is a set C of vertices that break G into 3 pieces A, B, and C where no edges connect A and B. We also add the requirement that A and B should be of roughly equal size.

Usually, we use edge partitioning for parallel sparse matrix-vector product and vertex partitioning for the ordering in direct sparse factorization.

18.3 Spectral Partitioning – One way to slice a problem in half

The three steps to illustrate the solution of that problem are:

A. Electrical Networks (for motiving the Laplace matrix of a graph)

B. Laplacian of a Graph

C. Partitioning (that uses the spectral information)

18.3.1 Electrical Networks

As an example we choose a certain configuration of resistors (1 ohm), which are combined as shown in fig. 18.1. A battery is connected to the nodes 3 and 4. It provides the voltage V_B. To obtain the current, we have to calculate the effective resistance of the configuration

$$I = \frac{V_B}{\text{eff. res.}}. \tag{18.1}$$

This so called *Graph* is by definition a *collection of nodes and edges.*

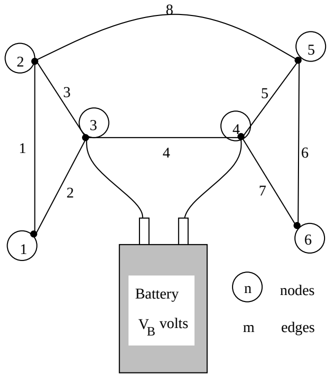


Figure 18.1: Resistor network with nodes and edges

18.3.2 Laplacian of a Graph

Kirchoff's Law tells us

$$\begin{pmatrix} 2 & -1 & -1 & & & \\ -1 & 3 & -1 & & -1 & \\ -1 & -1 & 3 & -1 & & \\ & & -1 & 3 & -1 & -1 \\ & -1 & & -1 & 3 & -1 \\ & & & -1 & -1 & 2 \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \\ V_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \tag{18.2}$$

The matrix contains the information of how the resistors are connected, it is called the *Laplacian of the Graph*

$$\nabla_G^2 = n \times n \quad \text{matrix,} \quad \begin{array}{l} \text{G...graph} \\ \text{n...nodes} \\ \text{m...edges} \end{array} \tag{18.3}$$

$$\left(\nabla_G^2\right)_{ii} = \text{degree of node } i \tag{18.4}$$

$$\left(\nabla_G^2\right)_{ij} = \begin{cases} 0 & \text{if } \nexists \text{ edge between node } i \text{ and node } j \\ -1 & \text{if } \exists \text{ edge between node } i \text{ and node } j \end{cases} \tag{18.5}$$

Yet there is another way to obtain the Laplacian. First we have to set up the so called *Node–edge Incidence matrix*, which is a $m \times n$ matrix and its elements are either 1, -1 or 0 depending on whether the edge (row of the matrix) connects the node (column of matrix) or not. We find

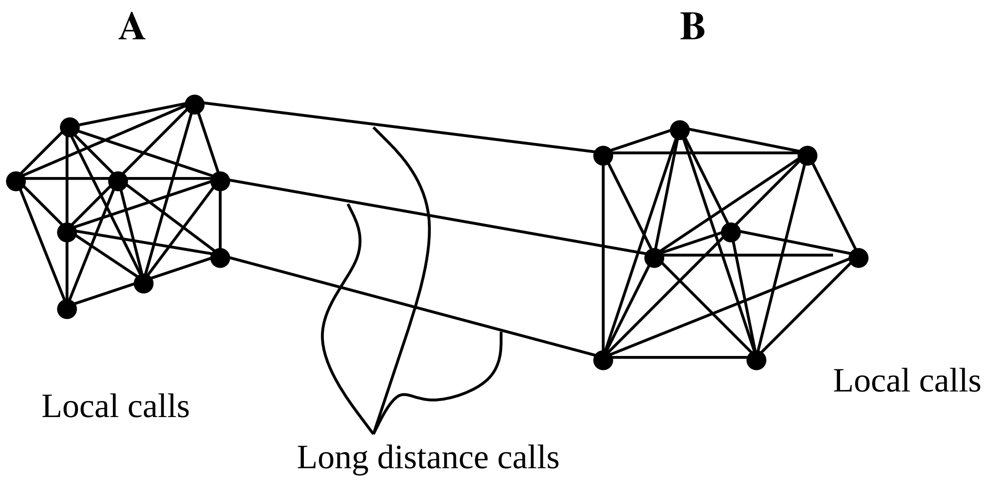


Figure 18.2: Partitioning of a telephone net as an example for a graph comparable in cost

$$\begin{array}{rcl} & \text{nodes} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \mathbf{M}_G \;= & \begin{array}{c} \text{edges} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} & \left(\begin{array}{cccccc} & & & & & \\ 1 & -1 & & & & \\ 1 & & -1 & & & \\ & 1 & -1 & & & \\ & & 1 & -1 & & \\ & & & 1 & -1 & \\ & & & & 1 & -1 \\ & & & 1 & & -1 \\ & 1 & & & -1 & \end{array}\right) \end{array} \qquad (18.6)$$

$$\mathbf{M}_G^T \mathbf{M}_G = \nabla_G^2 \qquad (18.7)$$

18.3.3 Spectral Partitioning

Partitioning means that we would like to break the problem into 2 parts A and B, whereas the number of connections between both parts are to be as small as possible (because they are the most expensive ones), see fig. 18.2. Now we define a vector $\mathbf{x} \in \mathcal{R}^n$ having the values $\mathbf{x}_i = \pm 1$. $+1$ stands for *belongs to part A*, -1 means *belongs to part B*. With use of some vector calculus we can show the following identity

$$ {}^{n}_{i=1} (\mathbf{M}_G \mathbf{x})_i^2 = (\mathbf{M}_G \mathbf{x})^T (\mathbf{M}_G \mathbf{x}) = \mathbf{x}^T \mathbf{M}_G^T \mathbf{M}_G \mathbf{x} = \mathbf{x}^T \nabla_G^2 \mathbf{x} \qquad (18.8)$$

$$\mathbf{x}^T \nabla_G^2 \mathbf{x} = 4 \times (\#\ \text{edges between A and B}).$$

In order to find the vector $\mathbf{x}$ with the least connections between part A and B, we want to solve the minimization problem:

$$\left.\begin{array}{r} \text{Min. } \mathbf{x}^T \nabla_G^2 \mathbf{x} \\ x_i = \pm 1 \\ x_i = 0 \end{array}\right\} \geq \left\{\begin{array}{r} \text{Min. } \mathbf{x}^T \nabla_G^2 \mathbf{x} \\ x_i \in \mathcal{R}^n \\ x_i^2 = n \\ x_i = 0 \end{array}\right. \qquad (18.9)$$

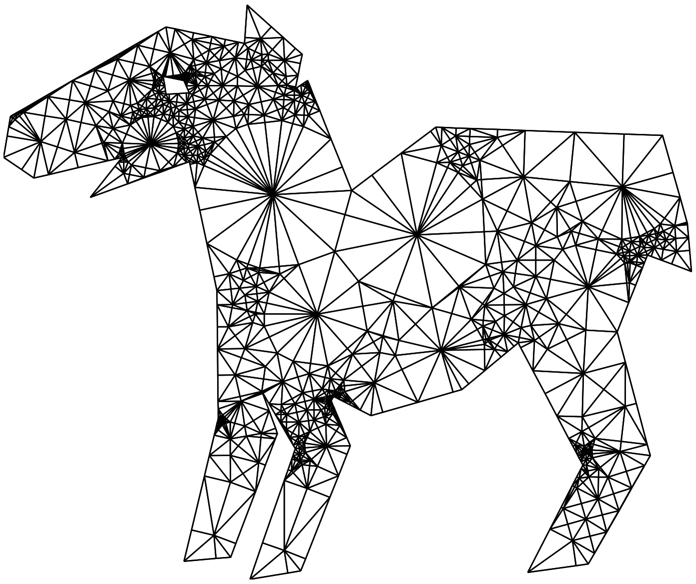


Figure 18.3: Tapir (Bern-Mitchell-Ruppert)

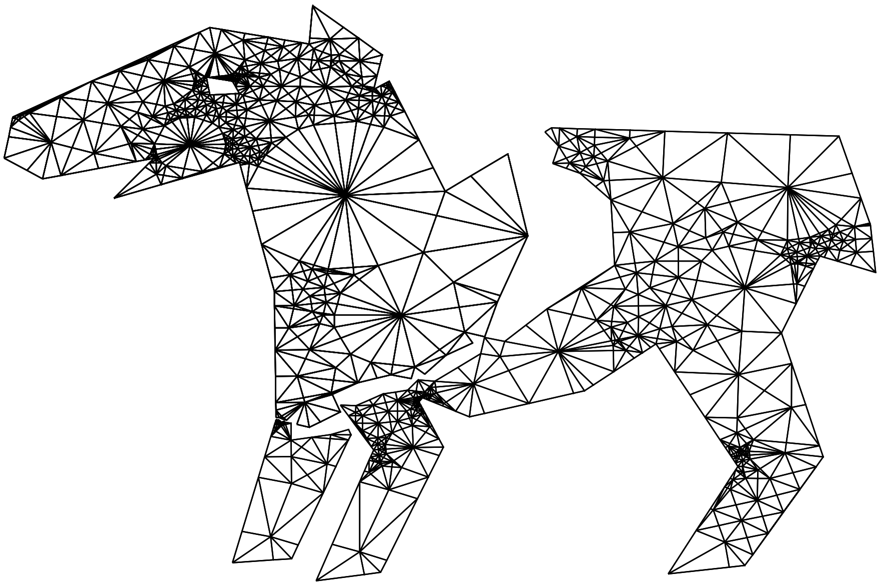
Finding an optimal ± 1 solution is intractable (NP-hard) in general. In practice, we relax the integer condition and allow the solution to be real. We do require the norm square of the solution, like the integer case, equal to n (see RHS of eqn. (18.9)). The relaxation enlarges the solution space, hence its optimal solution is no more than that of its integer counterpart. As an heuristic, we solve the relaxed version of the problem and "round" the solution to give an ± 1 solution.

The solution of the problem on the RHS of eqn. (18.9) gives us the second smallest eigenvalue of the Laplacian ∇_G^2 and the corresponding eigenvector, which is in fact the vector $\mathbf{x}$ with the least connections in the relaxation sense. To obtain a partition of the graph, we can divide its node set according to the vector x. We can even control the ratio of the partition. For example to obtain a bisection, i.e., a partition of each size, we can find the median of x and put those nodes whose x values is smaller than the median on one side and the remaining on the other side. We can also use the similar idea to divide the graph into two parts in which one is twice as big as the another.

Figure 18.3 shows a mesh together with its spectral partition. For those of you as ignorant as your professors, a tapir is defined by Webster as any of several large inoffensive chiefly nocturnal ungulates (family Tap iridae) of tropical America, Malaya, and Sumatra related to the horses and rhinoceroses. The tapir mesh is generated by a Matlab program written by Scott Mitchell based on a non-obtuse triangulation algorithm of Bern-Mitchell-Ruppert. The partition in the figure is generated by John Gilbert's spectral program in Matlab.

One has to notice that the above method is merely a heuristic. The algorithm does not come with any performance guarantee. In the worst case, the partition may be far from the optimal, partially due to the round-off effect and partially due the requirement of equal-sized partition. In fact the partition for the tapir graph is not quite optimal. Nevertheless, the spectral method, with the help of various improvements, works very well in practice. We will cover the partitioning problem more systematically later in the semester.

The most commonly used algorithm for finding the second eigenvalue and its eigenvector is the Lanczos algorithm though it makes far better sense to compute singular values rather than eigenvalues. Lanczos is an iterative algorithm that can converge quite quickly.



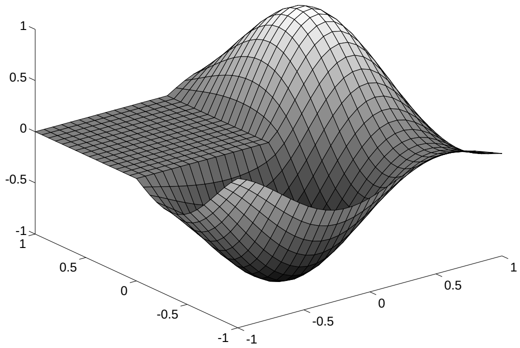


Figure 18.5: The 2nd eigenfunction of MathWork's logo

In general, we may need to divide a graph into more than one piece. The most commonly used approach is to recursively apply the partitioner that divides the graph into two pieces of roughly equal size. More systematic treatment of the partitioning problem will be covered in future lectures.

The success of the spectral method in practice has a physical interpretation. Suppose now we have a continuous domain in place of a graph. The Laplacian of the domain is the continuous counterpart of the Laplacian of a graph. The kth eigenfunction gives the kth mode of vibration and the second eigenfunction induce a cut (break) of the domain along the weakest part of the domain. (See figure18.5)

18.4 Geometric Methods

The geometric method developed by Miller, Teng, Thurston, and Vavasis guarantees to find a partitioning of a d-dimensional unstructured mesh of n of cut size $O(n^{1-1/d})$, a bound which is same as the the cut size for regular grid of the same size. Notice that such a bound does not hold for spectral method in general.

The partition of the graph into two pieces is defined by a circle (that is, a sphere in $\mathbb{R}^d$). The algorithm chooses the separating circle at random, from a distribution that is carefully constructed

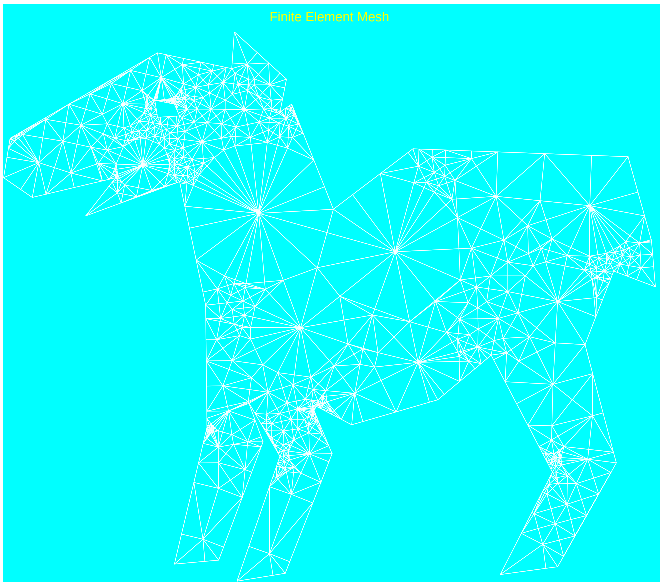


Figure 18.6: The input is a mesh with specified coordinates. Every triangle must be "well-shaped", which means that no angle can be too small. **Remarks:** The mesh is a subgraph of the intersection graph of a set of disks, one centered at each mesh point. Because the triangles are well-shaped, only a bounded number of disks can overlap any point, and the mesh is an "alpha-overlap graph". This implies that it has a good separator, which we proceed to find.

so that the separator will satisfy the $O(n^{1-1/d})$ bound with high probability. The distribution is described in terms of a stereographic projection and conformal mapping on the surface of a sphere of one dimension higher, in $\mathbb{R}^{d+1}$. To motivate the software development aspect of this approach, we use the following figures (Figures 18.6 – 18.13) generated by a Matlab implementation (written by Gilbert and Teng) to outline the steps for dividing a well-shaped mesh into two pieces. The algorithm works on meshes in any dimension, but we'll stick to two dimensions for the purpose of visualization.

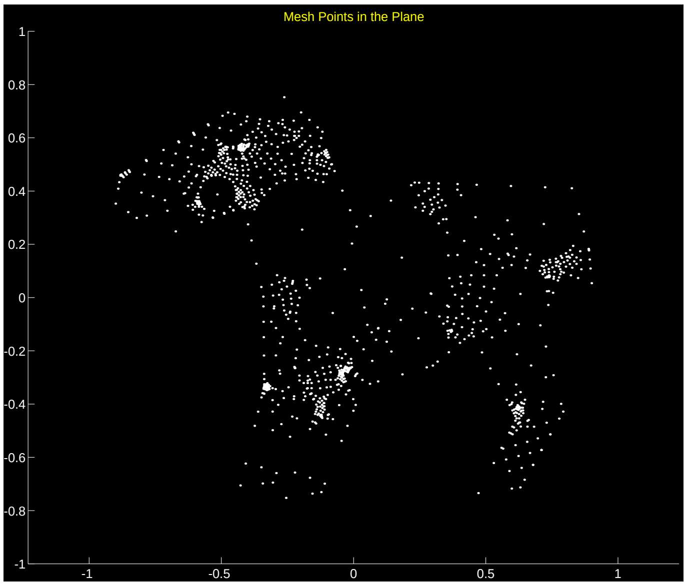


Figure 18.7: Let's redraw the mesh, omitting the edges for clarity.

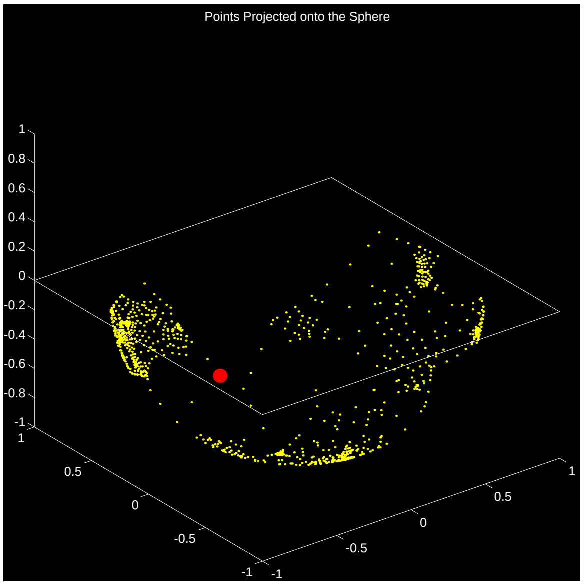


Figure 18.8: First we project the points stereographically from the plane onto a sphere (of one higher dimension than the mesh); Now we compute a "centerpoint" for the projected points in 3-space. Every plane through the centerpoint separates the input points into two roughly equal subsets. (Actually it's too expensive to compute a real centerpoint, so we use a fast, randomized heuristic to find a pretty good approximation.

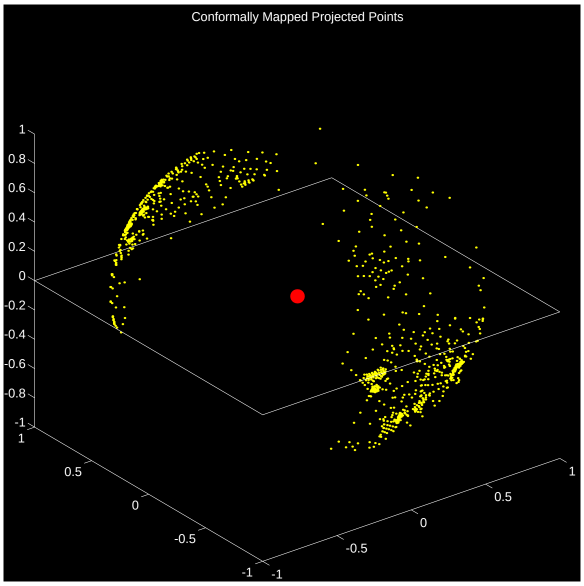


Figure 18.9: Next, we conformally map the points so that the centerpoint maps to the center of the sphere. This takes two steps: First we reflect the points on the sphere so the centerpoint is on the z axis, and then we scale the points in the plane to move the centerpoint along the z axis to the origin. The figures show the final result on the sphere and in the plane.

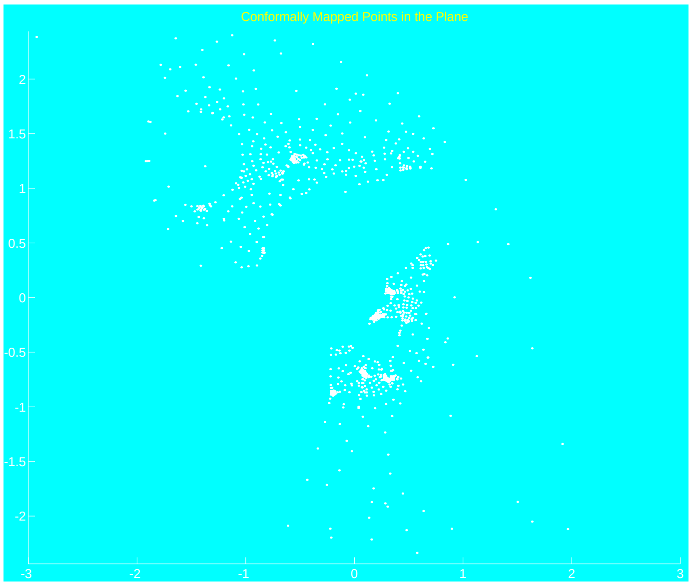


Figure 18.10: Any plane through the origin divides the points roughly evenly. Also, most planes only cut a small number of mesh edges (O(sqrt(n)), to be precise). Thus we find a separator by choosing a plane through the origin, which induces a great circle on the sphere. We shift the circle slightly to make the split exactly even. The second circle is the shifted version.

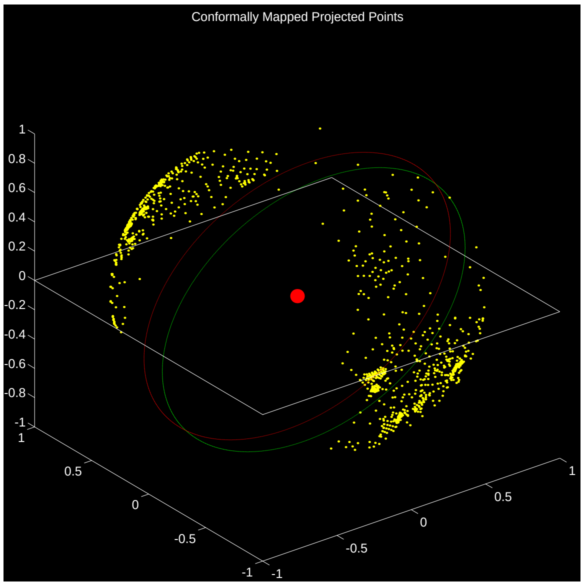


Figure 18.11: To translate this into a separator for the mesh, we first undo the conformal mapping, giving a (non-great) circle on the original sphere ...

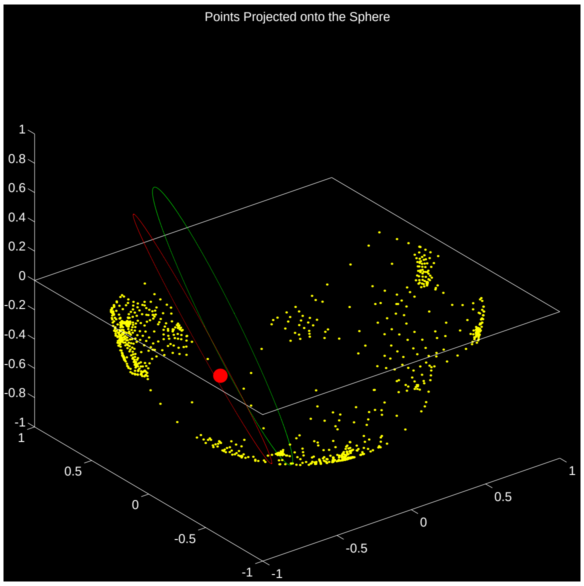


Figure 18.12: ... and then undo the stereographic projection, giving a circle in the original plane.

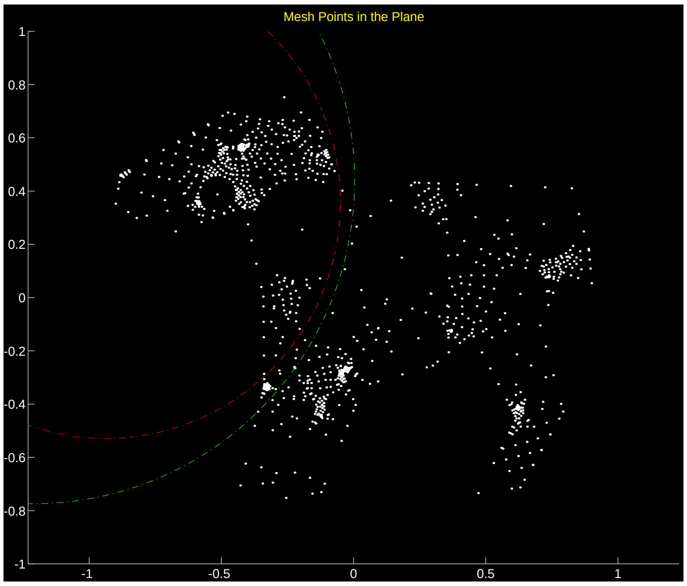


Figure 18.13: This partitions the mesh into two pieces with about n/2 points each, connected by at most O(sqrt(n)) edges. These connecting edges are called an "edge separator".

18.4.1 Geometric Graphs

This method applies to meshes in both two and three dimensions. It is based on the following important observation: graphs from large-scale problems in scientific computing are often defined geometrically. They are meshes of elements in a fixed dimension (typically two and three dimensions), that are *well shaped* in some sense, such as having elements of bounded aspect ratio or having elements with angles that are not too small. In other words, they are graphs embedded in two or three dimensions that come with natural geometric coordinates and with structures.

We now consider the types of graphs we expect to run these algorithms on. We don't expect to get a truly random graph. In fact, Erdös, Graham, and Szemerédi proved in the 1960s that with probability = 1, a random graph with cn edges does not have a two-way division with $o(n)$ crossing edges.

Structured graphs usually have divisions with $\sqrt{n}$ crossing edges. The following classes of graphs usually arise in applications such as finite element and finite difference methods (see Chapter **??**):

- **Regular Grids**: These arise, for example, from finite difference methods.
- **'Quad"-tree graphs and "Meshes"**: These arise, for example, from finite difference methods and hierarchical N-body simulation.
- **k-nearest neighbour graphs in d dimensions** Consider a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in $\mathbb{R}^d$. The vertex set for the graph is $\{1, 2, \ldots, n\}$ and the edge set is $\{(i, j)$: $p_j$is one of the k-nearest neighbours of p_i or vice-versa$\}$. This is an important class of graphs for image processing.
- **Disk packing graphs**: If a set of non-overlapping disks is laid out in a plane, we can tell which disks are touch. The nodes of a disk packing graph are the centers of the disks, and edges connect two nodes if their respective disks touch:

- **Planar graphs**: These are graphs that can be drawn in a plane without having crossing edges. Note that disk packing graphs are planar, and in fact every planar graph is isomorphic to some disk-packing graph (Andreev and Thurston).

18.4.2 Geometric Partitioning: Algorithm and Geometric Modeling

The main ingredient of the geometric approach is a novel geometrical characterization of graphs embedded in a fixed dimension that have a small *separator*, which is a relatively small subset of vertices whose removal divides the rest of the graph into two pieces of approximately equal size. By taking advantage of the underlying geometric structure, partitioning can be performed efficiently.

Computational meshes are often composed of elements that are *well-shaped* in some sense, such as having bounded aspect ratio or having angles that are not too small or too large. Miller et al. define a class of so-called *overlap graphs* to model this kind of geometric constraint.

An overlap graph starts with a *neighborhood system*, which is a set of closed disks in d-dimensional Euclidean space and a parameter k that restricts how deeply they can intersect.

Definition 18.4.1 *A k-*ply neighborhood system *in d* dimensions *is a set $\{D_1, \ldots, D_n\}$ of closed disks in $\mathbb{R}^d$, such that no point in $\mathbb{R}^d$ is strictly interior to more than k of the disks.*

A neighborhood system and another parameter α define an overlap graph. There is a vertex for each disk. For $\alpha = 1$, an edge joins two vertices whose disks intersect. For $\alpha > 1$, an edge joins two vertices if expanding the smaller of their two disks by a factor of α would make them intersect.

Definition 18.4.2 *Let $\alpha \geq 1$, and let $\{D_1, \ldots, D_n\}$ be a k-ply neighborhood system. The (α, k)-*overlap graph *for the neighborhood system is the graph with vertex set $\{1, \ldots, n\}$ and edge set*

$$\{(i,j) | (D_i \cap (\alpha \cdot D_j) \neq \emptyset) \text{ and } ((\alpha \cdot D_i) \cap D_j \neq \emptyset)\}.$$

We make an overlap graph into a mesh in d-space by locating each vertex at the center of its disk.

Overlap graphs are good models of computational meshes because every mesh of bounded-aspect-ratio elements in two or three dimensions is contained in some overlap graph (for suitable choices of the parameters α and k). Also, every planar graph is an overlap graph. Therefore, any theorem about partitioning overlap graphs implies a theorem about partitioning meshes of bounded aspect ratio and planar graphs.

We now describe the geometric partitioning algorithm.

We start with two preliminary concepts. We let Π denote the *stereographic projection* mapping from $\mathbb{R}^d$ to S^d, where S^d is the unit d-sphere embedded in $\mathbb{R}^{d+1}$. Geometrically, this map may be defined as follows. Given $\boldsymbol{x} \in \mathbb{R}^d$, append '0' as the final coordinate yielding $\boldsymbol{x}' \in \mathbb{R}^{d+1}$. Then compute the intersection of S^d with the line in $\mathbb{R}^{d+1}$ passing through $\boldsymbol{x}'$ and $(0, 0, \ldots, 0, 1)^T$. This intersection point is $\Pi(\boldsymbol{x})$.

Algebraically, the mapping is defined as

$$\Pi(\boldsymbol{x}) = \begin{pmatrix} 2\boldsymbol{x}/\chi \\ 1 - 2/\chi \end{pmatrix}$$

where $\chi = \boldsymbol{x}^T\boldsymbol{x} + 1$. It is also simple to write down a formula for the inverse of Π. Let $\boldsymbol{u}$ be a point on S^d. Then

$$\Pi^{-1}(\boldsymbol{u}) = \frac{\bar{\boldsymbol{u}}}{1 - u_{d+1}}$$

where $\bar{\boldsymbol{u}}$ denotes the first d entries of $\boldsymbol{u}$ and u_{d+1} is the last entry. The stereographic mapping, besides being easy to compute, has a number of important properties proved below.

A second crucial concept for our algorithm is the notion of a *center point*. Given a finite subset $P \subset \mathbb{R}^d$ such that $|P| = n$, a *center point* of P is defined to be a point $\boldsymbol{x} \in \mathbb{R}^d$ such that if H is any open halfspace whose boundary contains $\boldsymbol{x}$, then

$$|P \cap H| \leq dn/(d+1). \tag{18.10}$$

It can be shown from Helly's theorem [24] that a center point always exists. Note that center points are quite different from centroids. For example, a center point (which, in the $d = 1$ case, is the same as a median) is largely insensitive to "outliers" in P. On the hand, a single distant outlier can cause the centroid of P to be displaced by an arbitrarily large distance.

Geometric Partitioning Algorithm

Let $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ be the input points in $\mathbb{R}^d$ that define the overlap graph.

1. Given $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$, compute $P' = \{\Pi(\boldsymbol{p}_1), \ldots, \Pi(\boldsymbol{p}_n)\}$ so that $P' \subset S^d$.

2. Compute a center point $\boldsymbol{z}$ of P'.

3. Compute an orthogonal $(d+1) \times (d+1)$ matrix Q such that $Q\boldsymbol{z} = \boldsymbol{z}'$ where

$$\boldsymbol{z}' = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \theta \end{pmatrix}$$

   such that θ is a scalar.

4. Define $P'' = QP'$ (i.e., apply Q to each point in P'). Note that $P'' \subset S^d$, and the center point of P'' is $\boldsymbol{z}'$.

5. Let D be the matrix $[(1-\theta)/(1+\theta)]^{1/2} I$, where I is the $d \times d$ identity matrix. Let $P''' = \Pi(D\Pi^{-1}(P''))$. Below we show that the origin is a center point of P'''.

6. Choose a random great circle S_0 on S^d.

7. Transform S_0 back to a sphere $S \subset \mathbb{R}^d$ by reversing all the transformations above, i.e., $S = \Pi^{-1}(Q^{-1}\Pi(D^{-1}\Pi^{-1}(S_0)))$.

8. From S compute a set of vertices of G that split the graph as in Theorem **??**. In particular, define C to be vertices embedded "near" S, define A be vertices of $G - C$ embedded outside S, and define B to be vertices of $G - C$ embedded inside S.

We can immediately make the following observation: because the origin is a center point of P''', and the points are split by choosing a plane through the origin, then we know that $|A| \leq (d+1)n/(d+2)$ and $|B| \leq (d+1)n/(d+2)$ regardless of the details of how C is chosen. (Notice that the constant factor is $(d+1)/(d+2)$ rather than $d/(d+1)$ because the point set P' lies in $\mathbb{R}^{d+1}$ rather than $\mathbb{R}^d$.) Thus, one of the claims made in Theorem **??** will follow as soon as we have shown that the origin is indeed a center point of P''' at the end of this section.

We now provide additional details about the steps of the algorithm, and also its complexity analysis. We have already defined stereographic projection used Step 1. Step 1 requires $O(nd)$ operations.

Computing a true center point in Step 2 appears to a very expensive operation (involving a linear programming problem with n^d constraints) but by using random (geometric) sampling, an approximate center point can be found in random constant time (independent of n but exponential in d) [91, 45]. An *approximate* center point satisfies 18.10 except with $(d+1+\epsilon)n/(d+2)$ on the right-hand side, where $\epsilon > 0$ may be arbitrarily small. Alternatively, a deterministic linear-time sampling algorithm can be used in place of random sampling [61, 90], but one must again compute a center of the sample using linear programming in time exponential in d [63, 39].

In Step 3, the necessary orthogonal matrix may be represented as a single Householder reflection—see [40] for an explanation of how to pick an orthogonal matrix to zero out all but one entry in a vector. The number of floating point operations involved is $O(d)$ independent of n.

In Step 4 we do not actually need to compute P''; the set P'' is defined only for the purpose of analysis. Thus, Step 4 does not involve computation. Note that the $\boldsymbol{z}'$ is the center point of P'' after this transformation, because when a set of points is transformed by any orthogonal transformation, a center point moves according to the same transformation (more generally, center points are similarly moved under any affine transformation). This is proved below.

In Step 6 we choose a random great circle, which requires time $O(d)$. This is equivalent to choosing plane through the origin with a randomly selected orientation. (This step of the algorithm can be made deterministic; see [**?**].) Step 7 is also seen to require time $O(d)$.

Finally, there are two possible alternatives for carrying out Step 8. One alternative is that we are provided with the neighborhood system of the points (i.e., a list of n balls in $\mathbb{R}^d$) as part of the input. In this case Step 8 requires $O(nd)$ operations, and the test to determine which points belong in A, B or C is a simple geometric test involving S. Another possibility is that we are provided with the nodes of the graph and a list of edges. In this case we determine which nodes belong in A, B, or C based on scanning the adjacency list of each node, which requires time linear in the size of the graph.

Theorem 18.4.1 *If M is an unstructured mesh with bounded aspect ratio, then the graph of M is a subgraph of a bounded overlap graph of the neighborhood system where we have one ball for each vertex of M of radius equal to half of the distance to its nearest vertices. Clearly, this neighborhood system has ply equal to 1.*

Theorem 18.4.1 (Geometric Separators [63]) *Let G be an n-vertex (α, k)-overlap graph in d dimensions. Then the vertices of G can be partitioned into three sets A, B, and C, such that*

- *no edge joins A and B,*
- *A and B each have at most $(d+1)/(d+2)$ vertices,*
- *C has only $O(\alpha k^{1/d} n^{(d-1)/d})$ vertices.*

Such a partitioning can be computed by the geometric-partitioning-algorithm in randomized linear time sequentially, and in $O(n/p)$ parallel time when we use a parallel machine of p processors.

18.4.3 Other Graphs with small separators

The following classes of graphs all have small separators:

- Lines have edge-separators of size 1. Removing the middle edge is enough.
- Trees have a 1-vertex separator with $\beta = 2/3$ - the so-called centroid of the tree.

- Planar Graphs. A result of Lipton and Tarjan shows that a planar graph of bounded degree has a $\sqrt{8n}$ vertex separator with $\beta = 2/3$.
- d dimensional regular grids (those are used for basic finite difference method). As a folklore, they have a separator of size $n^{1-1/d}$ with beta $\beta = 1/2$.

18.4.4 Other Geometric Methods

Recursive coordinate bisection

The simplest form of geometric partitioning is recursive coordinate bisection (RCB) [86, 92]. In the RCB algorithm, the vertices of the graph are projected onto one of the coordinate axes, and the vertex set is partitioned around a hyperplane through the median of the projected coordinates. Each resulting subgraph is then partitioned along a different coordinate axis until the desired number of subgraphs is obtained.

Because of the simplicity of the algorithm, RCB is very quick and cheap, but the quality of the resultant separators can vary dramatically, depending on the embedding of the graph in $\mathbb{R}^d$. For example, consider a graph that is "+"-shaped. Clearly, the best (smallest) separator consists of the vertices lying along a diagonal cut through the center of the graph. RCB, however, will find the largest possible separators, in this case, planar cuts through the centers of the horizontal and vertical components of the graph.

Inertia-based slicing

Williams [92] noted that RCB had poor worst case performance, and suggested that it could be improved by slicing orthogonal to the principal axes of inertia, rather than orthogonal to the coordinate axes. Farhat and Lesoinne implemented and evaluated this heuristic for partitioning [31].

In three dimensions, let $v = (v_x, v_y, v_z)^t$ be the coordinates of vertex v in $\mathbb{R}^3$. Then the inertia matrix I of the vertices of a graph with respect to the origin is given by

$$I = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}$$

where,

$$I_{xx} =_{v \in V} v_y^2 + v_z^2, \quad I_{yy} =_{v \in V} v_x^2 + v_z^2, \quad I_{zz} =_{v \in V} v_x^2 + v_y^2$$

and, for $i, j \in \{x, y, z\}, i \neq j$,

$$I_{ij} = I_{ji} = -_{v \in V} v_i v_j$$

The eigenvectors of the inertia matrix are the principal axes of the vertex distribution. The eigenvalues of the inertia matrix are the principal moments of inertia. Together, the principal axes and principal moments of inertia define the inertia ellipse; the axes of the ellipse are the principal axes of inertia, and the axis lengths are the square roots of the corresponding principal moments. Physically, the size of the principal moments reflect how the mass of the system is distributed with respect to the corresponding axis - the larger the principal moment, the more mass is concentrated at a distance from the axis.

Let I_1, I_2, and I_3 denote the principal axes of inertia corresponding to the principal moments $\alpha_1 \leq \alpha_2 \leq \alpha_3$. Farhat and Lesoinne projected the vertex coordinates onto I_1, the axis about which the mass of the system is most tightly clustered, and partitioned using a planar cut through the median. This method typically yielded a good initial separator, but did not perform as well recursively on their test mesh - a regularly structured "T"-shape.

Farhat and Lesoinne did not present any results on the theoretical properties of the inertia slicing method. In fact, there are pathological cases in which the inertia method can be shown to yield a very poor separator. Consider, for example, a "+"-shape in which the horizontal bar is very wide and sparse, while the vertical bar is relatively narrow but dense. I_1 will be parallel to the horizontal axis, but a cut perpendicular to this axis through the median will yield a very large separator. A diagonal cut will yield the smallest separator, but will not be generated with this method.

Gremban, Miller, and Teng show how to use moment of inertia to improve the geometric partitioning algorithm.

18.4.5 Partitioning Software

- **Chaco** written by Bruce Hendrickson and Rob Leland. To get code send email to bahendr@cs.sandia.gov (Bruce Hendrickson).
- Matlab Mesh Partitioning Toolbox: written by Gilbert and Teng. It includes both edge and vertex separators, recursive bipartition, nested dissection ordering, visualizations and demos, and some sample meshes. The complete toolbox is available by anonymous ftp from machine `ftp.parc.xerox.com` as file `/pub/gilbert/meshpart.uu`.
- **Spectral code**: Pothen, Simon, and Liou.

18.5 Load-Balancing N-body Simulation for Non-uniform Particles

The discussion of Chapter **??** was focused on particles that are more or less uniformly distributed. However, in practical simulations, particles are usually not uniformly distributed. Particles may be highly clustered in some regions and relatively scattered in some other regions. Thus, the hierarchical tree is adaptively generated, with smaller box for regions of clustered particles. The computation and communication pattern of a hierarchical method becomes more complex and often is not known explicitly in advance.

18.5.1 Hierarchical Methods of Non-uniformly Distributed Particles

In this chapter, we use the following notion of non-uniformity: We say a point set $P = \{\boldsymbol{p}_1, ..., \boldsymbol{p}_n\}$ in d dimensions is *μ-non-uniform* if the hierarchical tree generated for P has height $\log_{2^d}(n/m) + \mu$. In other words, the ratio of the size of smallest leaf-box to the root-box is $1/2^{\log_{2^d}(n/m)+\mu}$. In practice, μ is less than 100.

The Barnes-Hut algorithm, as an algorithm, can be easily generalized to the non-uniform case. We describe a version of FMM for non-uniformly distributed particles. The method uses the box-box interaction. FMM tries to maximize the number of FLIPs among large boxes and also tries to FLIP between roughly equal sized boxes, a philosophy which can be described as: let parents do as much work as possible and then do the left-over work as much as possible before passing to the next generation. Let $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the set of child-boxes of the root-box of the hierarchical tree. FMM generates the set of all *interaction-pairs* of boxes by taking the union of Interaction-pair$(\boldsymbol{c}_i, \boldsymbol{c}_j)$ for all $1 \leq i < j \leq 2^d$, using the Interaction-Pair procedure defined below.

Procedure Interaction-Pair ($\boldsymbol{b}_1$, $\boldsymbol{b}_2$)

- If $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are β-well-separated, then $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ is an interaction-pair.
- Else, if both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are leaf-boxes, then particles in $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are near-field particles.
- Else, if both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are not leaf-boxes, without loss of generality, assuming that $\boldsymbol{b}_2$ is at least as large as $\boldsymbol{b}_1$ and letting $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the child-boxes of $\boldsymbol{b}_2$, then recursively decide interaction pair by calling: Interaction-Pair($\boldsymbol{b}_1$,$\boldsymbol{c}_i$) for all $1 \leq i \leq 2^d$.
- Else, if one of $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ is a leaf-box, without loss of generality, assuming that $\boldsymbol{b}_1$ is a leaf-box and letting $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the child-boxes of $\boldsymbol{b}_2$, then recursively decide interaction pairs by calling: Interaction-Pair($\boldsymbol{b}_1$,$\boldsymbol{c}_i$) for all $1 \leq i \leq 2^d$.

FMM for far-field calculation can then be defined as: for each interaction pair $(\boldsymbol{b}_1, \boldsymbol{b}_2)$, letting $\Phi_i^p()$ $(i = 1, 2)$ be the multipole-expansion of $\boldsymbol{b}_i$, flip $\Phi_1^p()$ to $\boldsymbol{b}_2$ and add to $\boldsymbol{b}_2$'s potential Taylor-expansion. Similarly, flip $\Phi_2^p()$ to $\boldsymbol{b}_1$ and add to $\boldsymbol{b}_1$'s potential Taylor-expansion. Then traverse down the hierarchical tree in a preordering, shift and add the potential Taylor-expansion of the parent box of a box to its own Taylor-expansion.

Note that FMM for uniformly distributed particles has a more direct description (see Chapter **??**).

18.5.2 The Communication Graph for N-Body Simulations

In order to efficiently implement an N-body method on a parallel machine, we need to understand its communication pattern, which can be described by a graph that characterizes the pattern of information exchange during the execution of the method. The communication graph is defined on basic computational elements of the method. The basic elements of hierarchical N-body methods are boxes and points, where points give the locations of particles and boxes are generated by the hierarchical method. Formally, the communication graph is an edge-weighted directed graph, where the edges describe the pattern of communication and the weight on an edge specifies the communication requirement along the edge.

A Refined FMM for Non-Uniform Distributions

For parallel implementation, it is desirable to have a communication graph that uses small edge-weights and has small in- and out-degrees. However, some boxes in the set of interaction-pairs defined in the last section may have large degree!

FMM described in the last subsection has a drawback which can be illustrated by the following 2D example. Suppose the root-box is divided into four child-boxes A, B, C, and D. Assume further that boxes A, B and C contains less than m (< 100) particles, and most particles, say n of them, are uniformly distributed in D, see Figure 18.14. In FMM, we further recursively divide D by $\log_4(n/m)$ levels. Notice that A, B, and C are not well-separated from any box in D. Hence the FMM described in the previous subsection will declare all particles of D as near-field particles of A, B, and C (and vice versa). The drawback is two-folds: (1) From the computation viewpoint, we cannot take advantage of the hierarchical tree of D to evaluate potentials in A, B, and C. (2) From the communication viewpoint, boxes A, B, and C have a large in-degree in the sense that each particle in these boxes need to receive information from all n particles in D, making partitioning and load balancing harder. Notice that in BH most boxes of D are well-separated from particles in A, B, and C. Hence the well-separation condition is different in BH: *because BH uses the particle-box interaction, the well-separation condition is measured with respect to the size of the boxes in*

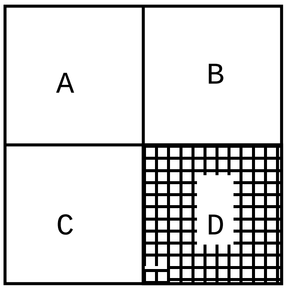


Figure 18.14: A non-uniform example

D. Thus most boxes are well-separated from particles in A, B, and C. In contrast, because FMM applies the FLIP operation, the well-separation condition must measure up against the size of the larger box. Hence no box in D is well-separated from A, B, and C.

Our refined FMM circumvents this problem by incorporating the well-separation condition of BH into the Interaction-Pair procedure: if $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are not well-separated, and $\boldsymbol{b}_1$, the larger of the two, is a leaf-box, then we use a well-separation condition with respect to $\boldsymbol{b}_2$, instead of to $\boldsymbol{b}_1$, and apply the FLIP operation directly onto particles in the leaf-box $\boldsymbol{b}_1$ rather than $\boldsymbol{b}_1$ itself.

We will define this new well-separation condition shortly. First, we make the following observation about the Interaction-Pair procedure defined in the last subsection. We can prove, by a simple induction, the following fact: if $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are an interaction-pair and both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are not leaf-boxes, then $1/2 \leq size(\boldsymbol{b}_1)/size(\boldsymbol{b}_2) \leq 2$. This is precisely the condition that FMM would like to maintain. For uniformly distributed particles, such condition is always true between any interaction-pair (even if one of them is a leaf-box). However, for non-uniformly distributed particles, if $\boldsymbol{b}_1$, the larger box, is a leaf-box, then $\boldsymbol{b}_1$ could be much larger than $\boldsymbol{b}_2$.

The new *β-well-separation* condition, when $\boldsymbol{b}_1$ is a leaf-box, is then defined as: $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are *β-well-separated* if $\boldsymbol{b}_2$ is well-separated from all particles of $\boldsymbol{b}_1$ (as in BH). Notice, however, with the new condition, we can no longer FLIP the multipole expansion of $\boldsymbol{b}_1$ to a Taylor-expansion for $\boldsymbol{b}_2$. Because $\boldsymbol{b}_1$ has only a constant number of particles, we can directly evaluate the potential induced by these particles for $\boldsymbol{b}_2$. This new condition makes the FLIP operation of this special class of interaction-pairs *uni-directional*: We only FLIP $\boldsymbol{b}_2$ to $\boldsymbol{b}_1$.

We can describe the refined Interaction-Pair procedure using modified well-separation condition when one box is a leaf-box.

Procedure Refined Interaction-Pair ($\boldsymbol{b}_1$, $\boldsymbol{b}_2$)

- If $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are β-well-separated and $1/2 \leq size(\boldsymbol{b}_1)/size(\boldsymbol{b}_2) \leq 2$, then $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ is a bi-directional interaction-pair.

- Else, if the larger box, without loss of generality, $\boldsymbol{b}_1$, is a leaf-box, then the well-separation condition becomes: $\boldsymbol{b}_2$ is well-separated from all particles of $\boldsymbol{b}_1$. If this condition is true, then $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ is a uni-directional interaction-pair from $\boldsymbol{b}_2$ to $\boldsymbol{b}_1$.

- Else, if both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are leaf-boxes, then particles in $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are near-field particles.

- Else, if both $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are not leaf-boxes, without loss of generality, assuming that $\boldsymbol{b}_2$ is at least as large as $\boldsymbol{b}_1$ and letting $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the child-boxes of $\boldsymbol{b}_2$, then recursively decide interaction-pairs by calling: Interaction-Pair($\boldsymbol{b}_1$,$\boldsymbol{c}_i$) for all $1 \leq i \leq 2^d$.

- Else, if one of $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ is a leaf-box, without loss of generality, assuming that $\boldsymbol{b}_1$ is a leaf box and letting $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the child-boxes of $\boldsymbol{b}_2$, then recursively decide interaction pairs by calling: Interaction-Pair($\boldsymbol{b}_1$,$\boldsymbol{c}_i$) for all $1 \leq i \leq 2^d$.

Let $\boldsymbol{c}_1, ..., \boldsymbol{c}_{2^d}$ be the set of child-boxes of the root-box of the hierarchical tree. Then the set of all interaction-pair can be generated as the union of Refined-Interaction-Pair($\boldsymbol{c}_i, \boldsymbol{c}_j$) for all $1 \leq i < j \leq 2^d$.

The refined FMM for far-field calculation can then be defined as: for each bi-directional interaction pair $(\boldsymbol{b}_1, \boldsymbol{b}_2)$, letting $\Phi_i^p()$ $(i = 1, 2)$ be the multipole expansion of $\boldsymbol{b}_i$, flip $\Phi_1^p()$ to $\boldsymbol{b}_2$ and add to $\boldsymbol{b}_2$'s potential Taylor-expansion. Similarly, flip $\Phi_2^p()$ to $\boldsymbol{b}_1$ and add to $\boldsymbol{b}_1$'s potential Taylor-expansion. Then traverse down the hierarchical tree in a preordering, shift and add the potential Taylor-expansion of the parent box of a box to its own Taylor-expansion. For each uni-directional interaction pair $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ from $\boldsymbol{b}_2$ to $\boldsymbol{b}_1$, letting $\Phi_2^p()$ be the multipole-expansion of $\boldsymbol{b}_2$, evaluate $\Phi_2^p()$ directly for each particle in $\boldsymbol{b}_2$ and add its potential.

Hierarchical Neighboring Graphs

Hierarchical methods (BH and FMM) explicitly use two graphs: the *hierarchical tree* which connects each box to its parent box and each particle to its leaf-box, and the *near-field graph* which connects each box with its near-field boxes. The hierarchical tree is generated and used in the first step to compute the multipole expansion induced by each box. We can use a bottom-up procedure to compute these multipole expansions: First compute the multipole expansions at leaf-boxes and then *SHIFT* the expansion to the parent boxes and then up the hierarchical tree until multipole-expansions for all boxes in the hierarchical tree are computed.

The near-field graph can also be generated by the Refined-Interaction-Pair procedure. In Section 18.5.3, we will formally define the near-field graph.

Fast-Multipole Graphs (FM)

The Fast-Multipole graph, FM^β, models the communication pattern of the refined FMM. It is a graph defined on the set of boxes and particles in the hierarchical tree. Two boxes $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are connected in FM^β iff (1) $\boldsymbol{b}_1$ is the parent box of $\boldsymbol{b}_2$, or vice versa, in the hierarchical tree; or (2) $(\boldsymbol{b}_1, \boldsymbol{b}_2)$ is an interaction-pair generated by Refined-Interaction-Pair defined in Section 18.5.2. The edge is bi-directional for a bi-directional interaction-pair and uni-directional for a uni-directional interaction-pair. Furthermore, each particle is connected with the box that contains the particle.

The following Lemma that will be useful in the next section.

Lemma 18.5.1 *The refined FMM flips the multipole expansion of $\boldsymbol{b}_2$ to $\boldsymbol{b}_1$ if and only if (1) $\boldsymbol{b}_2$ is well-separated from $\boldsymbol{b}_1$ and (2) neither the parent of $\boldsymbol{b}_2$ is well-separated from $\boldsymbol{b}_1$ nor $\boldsymbol{b}_2$ is well-separated from the parent of $\boldsymbol{b}_1$.*

It can be shown that both in- and out-degrees of FM^β are small.

Barnes-Hut Graphs (BH)

BH defines two classes of communication graphs: BH_S^β and BH_P^β. BH_S^β models the sequential communication pattern and BH_P^β is more suitable for parallel implementation. The letters S and P, in BH_S^β and BH_P^β, respectively, stand for "Sequential" and "Parallel".

We first define BH_S^β and show why parallel computing requires a different communication graph BH_P^β to reduce total communication cost.

The graph BH_S^β of a set of particles P contains two sets of vertices: P, the particles, and B, the set of boxes in the hierarchical tree. The edge set of the graph BH_S^β is defined by the communication pattern of the sequential BH. A particle $\boldsymbol{p}$ is connected with a box $\boldsymbol{b}$ if in BH, we need to evaluate $\boldsymbol{p}$ against $\boldsymbol{b}$ to compute the force or potential exerted on $\boldsymbol{p}$. So the edge is directed from $\boldsymbol{b}$ to $\boldsymbol{p}$. Notice that if $\boldsymbol{p}$ is connected with $\boldsymbol{b}$, then $\boldsymbol{b}$ must be well-separated from $\boldsymbol{p}$. Moreover, the parent of $\boldsymbol{b}$ is not well-separated from $\boldsymbol{p}$. Therefore, if $\boldsymbol{p}$ is connected with $\boldsymbol{b}$ in BH_S^β, then $\boldsymbol{p}$ is not connected to any box in the subtree of $\boldsymbol{b}$ nor to any ancestor of $\boldsymbol{b}$.

In addition, each box is connected directly with its parent box in the hierarchical tree and each point $\boldsymbol{p}$ is connected its leaf-box. Both types of edges are bi-directional.

Lemma 18.5.2 *Each particle is connected to at most $O(\log n + \mu)$ number of boxes. So the in-degree of BH_S^β is bounded by $O(\log n + \mu)$.*

Notice, however, BH_S^β is not suitable for parallel implementation. It has a large out-degree. This major drawback can be illustrated by the example of n uniformly distributed particles in two dimensions. Assume we have four processors. Then the "best" way to partition the problem is to divide the root-box into four boxes and map each box onto a processor. Notice that in the direct parallel implementation of BH, as modeled by BH_S^β, each particle needs to access the information of at least one boxes in each of the other processors. Because each processor has $n/4$ particles, the total communication overhead is $\Omega(n)$, which is very expensive.

The main problem with BH_S^β is that many particles from a processor need to access the information of the same box in some other processors (which contributes to the large out-degree). We show that a combination technique can be used to reduce the out-degree. The idea is to *combine* the "same" information from a box and send the information as one unit to another box on a processor that needs the information. We will show that this combination technique reduces the total communication cost to $O(\sqrt{n \log n})$ for the four processor example, and to $O(\sqrt{pn \log n})$ for p processors. Similarly, in three dimensions, the combination technique reduces the volume of messages from $\Omega(n \log n)$ to $O(p^{1/3} n^{2/3} (\log n)^{1/3})$.

We can define a graph BH_P^β to model the communication and computation pattern that uses this combination technique. Our definition of BH_P^β is inspired by the communication pattern of the refined FMM. It can be shown that the communication pattern of the refined FMM can be used to guide the message combination for the parallel implementation of the Barnes-Hut method!

The combination technique is based on the following observation: Suppose $\boldsymbol{p}$ is well-separated from $\boldsymbol{b}_1$ but not from the parent of $\boldsymbol{b}_1$. Let $\boldsymbol{b}$ be the largest box that contains $\boldsymbol{p}$ such that $\boldsymbol{b}$ is well-separated from $\boldsymbol{b}_1$, using the well-separation definition in Section 18.5.2. If $\boldsymbol{b}$ is not a leaf-box, then $(\boldsymbol{b}, \boldsymbol{b}_1)$ is a bi-directional interaction-pair in the refined FMM. If $\boldsymbol{b}$ is a leaf-box, then $(\boldsymbol{b}, \boldsymbol{b}_1)$ is a uni-directional interaction-pair from $\boldsymbol{b}_1$ to $\boldsymbol{b}$. Hence $(\boldsymbol{b}, \boldsymbol{b}_1)$ is an edge of FM^β. Then, any other particle $\boldsymbol{q}$ contained in $\boldsymbol{b}$ is well-separated from $\boldsymbol{b}_1$ as well. Hence we can combine the information from $\boldsymbol{b}_1$ to $\boldsymbol{p}$ and $\boldsymbol{q}$ and all other particles in $\boldsymbol{b}$ as follows: $\boldsymbol{b}_1$ sends its information (just one copy) to $\boldsymbol{b}$ and $\boldsymbol{b}$ forwards the information down the hierarchical tree, to both $\boldsymbol{p}$ and $\boldsymbol{q}$ and all other particles in $\boldsymbol{b}$. This combination-based-communication scheme defines a new communication graph BH_P^β for parallel BH: The nodes of the graph are the union of particles and boxes, i.e., $P \cup B(P)$. Each particle is connected to the leaf-box it belongs to. Two boxes are connected iff they are connected in the Fast-Multipole graph. However, to model the communication cost, we must introduce a weight on each edge along the hierarchical tree embedded in BH_P^β, to be equal to the number of data units needed to be sent along that edge.

Lemma 18.5.3 *The weight on each edge in BH_P^β is at most $O(\log n + \mu)$.*

It is worthwhile to point out the difference between the comparison and communication patterns in BH. In the sequential version of BH, if $\boldsymbol{p}$ is connected with $\boldsymbol{b}$, then we have to compare $\boldsymbol{p}$ against all ancestors of $\boldsymbol{b}$ in the computation. The procedure is to first compare $\boldsymbol{p}$ with the root of the hierarchical tree, and then recursively move the comparison down the tree: if the current box compared is not well-separated from $\boldsymbol{p}$, then we will compare $\boldsymbol{p}$ against all its child-boxes. However, in terms of force and potential calculation, we only evaluate a particle against the first box down a path that is well-separated from the particle. The graphs BH_S^β and BH_P^β capture the communication pattern, rather than the comparison pattern. The communication is more essential to force or potential calculation. The construction of the communication graph has been one of the bottlenecks in load balancing BH and FMM on a parallel machine.

18.5.3 Near-Field Graphs

The near-field graph is defined over all leaf-boxes. A leaf-box $\boldsymbol{b}_1$ is a *near-field neighbor* of a leaf-box $\boldsymbol{b}$ if $\boldsymbol{b}_1$ is not well-separated from some particles of $\boldsymbol{b}$. Thus, FMM and BH directly compute the potential at particles in $\boldsymbol{b}$ induced by particles of $\boldsymbol{b}_1$.

There are two basic cases: (1) if $size(\boldsymbol{b}_1) \leq size(\boldsymbol{b})$, then we call $\boldsymbol{b}_1$ a *geometric near-field neighbor* of $\boldsymbol{b}$. (2) if $size(\boldsymbol{b}_1) > size(\boldsymbol{b})$, then we call $\boldsymbol{b}_1$ a *hierarchical near-field neighbor* of $\boldsymbol{b}$. In the example of Section 18.5.2, A, B, C are hierarchical near-field neighbors of all leaf-boxes in D; while A, B, and C have some geometric near-field neighbors in D.

We introduce some notations. The *geometric in-degree* of a box $\boldsymbol{b}$ is the number of its geometric near-field neighbors. The *geometric out-degree* of a box $\boldsymbol{b}$ is the number of boxes to which b is the geometric near-field neighbors. The *hierarchical in-degree* of a box $\boldsymbol{b}$ is the number of its hierarchical near-field neighbors. We will define the *hierarchical out-degree* of a box shortly.

It can be shown that the geometric in-degree, geometric out-degree, and hierarchical in-degree are small. However, in the example of Section 18.5.2, A, B, and C are hierarchical near-field neighbors for all leaf-boxes in D. Hence the number of leaf-boxes to which a box is a hierarchical near-field neighbor could be very large. So the near-field graph defined above can have a very large out-degree.

We can use the combination technique to reduce the degree when a box $\boldsymbol{b}$ is a hierarchical near-field neighbor of a box $\boldsymbol{b}_1$. Let $\boldsymbol{b}_2$ be the ancestor of $\boldsymbol{b}_1$ of the same size as $\boldsymbol{b}$. Instead of $\boldsymbol{b}$ sending its information directly to $\boldsymbol{b}_1$, $\boldsymbol{b}$ sends it to $\boldsymbol{b}_2$ and $\boldsymbol{b}_2$ then forwards the information down the hierarchical tree. Notice that $\boldsymbol{b}$ and $\boldsymbol{b}_2$ are not well-separated. We will refer to this modified near-field graph as the near-field graph, denoted by NF^β. We also define the *hierarchical out-degree* of a box $\boldsymbol{b}$ to be the number of edges from $\boldsymbol{b}$ to the set of non-leaf-boxes constructed above. We can show that the *hierarchical out-degree* is also small.

To model the near-field communication, similar to our approach for BH, we introduce a weight on the edges of the hierarchical tree.

Lemma 18.5.4 *The weight on each edge in NF^β is at most $O(\log n + \mu)$.*

18.5.4 N-body Communication Graphs

By abusing notations, let $FM^\beta = FM^\beta \cup NF^\beta$ and $BH_P^\beta = BH_P^\beta \cup NF^\beta$. So the communication graph we defined simultaneously supports near-field and far-field communication, as well as communication up and down the hierarchical tree. Hence by partitioning and load balancing FM^β

and BH_P^β , we automatically partition and balance the hierarchical tree, the near-field graph, and the far-field graph.

18.5.5 Geometric Modeling of N-body Graphs

Similar to well-shaped meshes, there is a geometric characterization of N-body communication graphs. Instead of using neighborhood systems, we use box-systems. A *box-system* in $\mathbb{R}^d$ is a set $B = \{B_1, \ldots, B_n\}$ of boxes. Let $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ be the centers of the boxes, respectively. For each integer k, the set B is a *k-ply box-system* if no point $\boldsymbol{p} \in \mathbb{R}^d$ is contained in more than k of $\mathrm{int}(B_1), \ldots, \mathrm{int}(B_n)$.

For example, the set of all leaf-boxes of a hierarchical tree forms a 1-ply box-system. The box-system is a variant of neighborhood system of Miller, Teng, Thurston, and Vavasis [63], where a neighborhood system is a collection of Euclidean balls in $\mathbb{R}^d$. We can show that box-systems can be used to model the communication graphs for parallel adaptive N-body simulation.

Given a box-system, it is possible to define the *overlap* graph associated with the system:

Definition 18.5.1 *Let $\alpha \geq 1$ be given, and let $\{B_1, \ldots, B_n\}$ be a k-ply box-system. The α-***overlap graph** *for this box-system is the undirected graph with vertices $V = \{1, \ldots, n\}$ and edges*

$$E = \{(i,j) : B_i \cap (\alpha \cdot B_j) \neq \emptyset \text{ and } (\alpha \cdot B_i) \cap B_j \neq \emptyset\}.$$

The edge condition is equivalent to: $(i,j) \in E$ iff the α dilation of the smaller box touches the larger box.

As shown in [90], the partitioning algorithm and theorem of Miller *et al* can be extended to overlap graphs on box-systems.

Theorem 18.5.1 *Let G be an α-overlap graph over a k-ply box-system in $\mathbb{R}^d$, then G can be partitioned into two equal sized subgraphs by removing at most $O(\alpha k^{1/d} n^{1-1/d})$ vertices. Moreover, such a partitioning can be computed in linear time sequentially and in parallel $O(n/p)$ time with p processors.*

The key observation is the following theorem.

Theorem 18.5.2 *Let $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ be a point set in $\mathbb{R}^d$ that is μ-non-uniform. Then the set of boxes $B(P)$ of hierarchical tree of P is a $(\log_{2^d} n + \mu)$-ply box-system and $FM^\beta(P)$ and $BH_P^\beta(P)$ are subgraphs of the 3β-overlap graph of $B(P)$.*

Therefore,

Theorem 18.5.3 *Let G be an N-body communication graph (either for BH or FMM) of a set of particles located at $P = \{\boldsymbol{p}_1, ..., \boldsymbol{p}_n\}$ in $\mathbb{R}^d$ ($d = 2$ or 3). If P is μ-non-uniform, then G can be partitioned into two equal sized subgraphs by removing at most $O(n^{1-1/d}(\log n + \mu)^{1/d})$ nodes. Moreover, such a partitioning can be computed in linear time sequentially and in parallel $O(n/p)$ time with p processors.*

Chapter 19

Mesh Generation

An essential step in scientific computing is to find a proper discretization of a continuous domain. This is the problem of *mesh generation*. Once we have a discretization or sometimes we just say a "mesh", differential equations for flow, waves, and heat distribution are then approximated by finite difference or finite element formulations. However, not all meshes are equally good numerically. Discretization errors depend on the geometric shape and size of the elements while the computational complexity for finding the numerical solution depends on the number of elements in the mesh and often the overall geometric quality of the mesh as well.

The most general and versatile mesh is an unstructured triangular mesh. Such a mesh is simply a triangulation of the input domain (e.g., a polygon), along with some extra vertices, called *Steiner points*. A triangulation is a decomposition of a space into a collection of interior disjoint simplices so that two simplices can only intersect at a lower dimensional simplex. We all know that in two dimensions, a simplex is a triangle and in three dimensions a simplex is a tetrahedron. A triangulation can be obtained by triangulating a point set, that form the vertices of the triangulation.

Even among triangulations some are better than others. Numerical errors depend on the *quality* of the triangulation, meaning the shapes and sizes of triangles.

In order for a mesh to be useful in approximating partial differential equations, it is necessary that discrete functions generated from the mesh (such as the piecewise linear functions) be capable of approximating the solutions sought. Classical finite element theory [17] shows that a sufficient condition for optimal approximation results to hold is that the minimum angle or the aspect ratio of each simplex in the triangulation be bounded independently of the mesh used; however, Babuska [4] shows that while this is sufficient, it is not a necessary condition. See Figure **??** for a triangulation whose minimum degree is at least 20 degree.

Automatic mesh generation is a relatively new field. No mathematically sound procedure for obtaining the 'best' mesh distribution is available. The criteria are usually heuristic.

The input description of physical domain has two components: the geometric definition of the domain and the numerical requirements within the domain. The geometric definition provides the boundary of the domain either in the form of a continuous model or of a discretized boundary model. Numerical requirements within the domain are typically obtained from an initial numerical simulation on a preliminary set of points. The numerical requirements obtained from the initial point set define an additional local spacing function restricting the final point set.

An automatic mesh generator try to generate an additional points to the internally and boundary of the domain to smooth out the mesh generation and concentrate mesh density where necessary - to optimize the total number of mesh points.

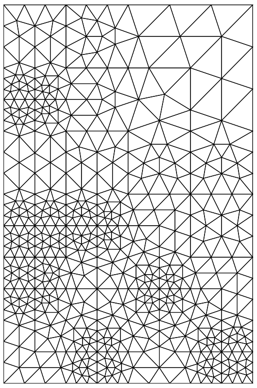


Figure 19.1: A well-shaped triangulation

19.1 How to Describe a Domain?

The most intuitive and obvious structure of a domain (for modeling a scientific problem) is its geometry.

One way to describe the geometry is to use *constructive solid geometry* formula. In this approach, we have a set of basic geometric primitive shapes, such as boxes, spheres, half-spaces, triangles, tetrahedra, ellipsoids, polygons, etc. We then define (or approximate) a domain as finite unions, intersections, differences, and complementation of primitive shapes, i.e., by a well-structured formula of a finite length of primitive shapes with operators that include union, intersection, difference, and complementation.

An alternative way is to discretize the boundary of the domain, and describes the domain as a polygonal (polyhedral) object (perhaps with holes). Often we convert the constructive solid geometry formula into the discretized boundary description for mesh generation.

For many computational applications, often, some other information of a domain and the problem are equally important for quality mesh generation.

The numerical spacing functions, typically denoted by $h(\boldsymbol{x})$, is usually defined at a point $\boldsymbol{x}$ by the eigenvalues of the Hessian matrix of the solution u to the governing partial differential equations (PDEs) [4, 89, 65]. Locally, u behaves like a quadratic function

$$u(\boldsymbol{x} + d\boldsymbol{x}) = \frac{1}{2}(\boldsymbol{x}H\boldsymbol{x}^T) + \boldsymbol{x}\nabla u(\boldsymbol{x}) + u(\boldsymbol{x}),$$

where H is the *Hessian matrix* of u, the matrix of second partial derivatives. The spacing of mesh points, required by the accuracy of the discretization at a point $\boldsymbol{x}$, is denoted by $h(\boldsymbol{x})$ and should depend on the reciprocal of the square root of the largest eigenvalues of H at $\boldsymbol{x}$.

When solving a PDE numerically, we estimate the eigenvalues of Hessian at a certain set of points in the domain based on the numerical approximation of the previous iteration [4, 89]. We then expand the spacing requirement induced by Hessian at these points over the entire domain.

For a problem with a smooth change in solution, we can use a (more-or-less) uniform mesh where all elements are of roughly equal size. On the other hand, for problem with rapid change in solution, such as earthquake, wave, shock modeling, we may use much dense grinding in the area of with high intensity. See Fig 19.2.So, the information about the solution structure can be of a great value to quality mesh generation.

Other type of information may come in the process of solving a simulation problem. For example, in adaptive methods, we may start with a much coarse and uniform grid. We then estimate the

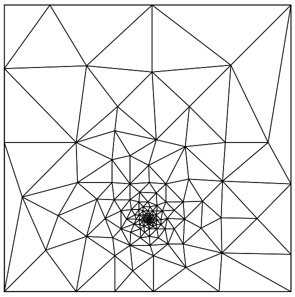


Figure 19.2: Triangulation of well-spaced point set around a singularity

error of the previous step. Based on the error bound, we then adaptively refine the mesh, e.g., make the area with larger error much more dense for the next step calculation. As we shall argue later, unstructured mesh generation is more about finding the proper distribution of mesh point then the discretization itself (this is a very personal opinion).

19.2 Types of Meshes

- **Structured grids** divide the domain into regular grid squares. For examples finite difference gridding on a square grid. Matrices based on structured grids are very easy and fast to assemble. Structured grids are easy to generate; numerical formulation and solution based on structured grids are also relatively simple.

- **Unstructured grids** decompose the domain into simple mesh elements such as simplices based on a density function that is defined by the input geometry or the numerical requirements (e.g., from error estimation). But the associated matrices are harder and slower to assemble compared to the previous method; the resulting linear systems are also relatively hard to solve. Most of finite element meshes used in practice are of the unstructured type.

- **Hybrid grids** are generated by first decomposing the domain into non-regular domain and then decomposing each such domain by a regular grid. Hybrid grids are often used in domain decomposition.

Structured grids are much easy to generate and manipulate. The numerical theory of this discretization is better understood. However, its applications is limited to problems with simple domain and smooth changes in solution. For problems with complex geometry whose solution changes rapidly, we need to use an *unstructured mesh* to reduce the problem size. For example, when modeling earthquake we want a dense discretization near the quake center and a sparse discretization in the regions with low activities. It would be waste to give regions with low activities as fine a discretization as the regions with high activities. Unstructured meshes are especially important for three dimensional problems.

The adaptability of unstructured meshes comes with new challenges, especially for 3D problems. However, the numerical theory becomes more difficult – this is an outstanding direction for future research; the algorithmic design becomes much harder.

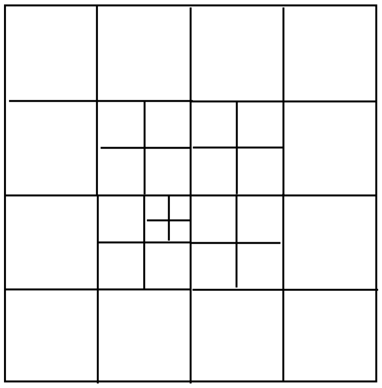


Figure 19.3: A quadtree

19.3 Refinement Methods

A mesh generator usually does two things: (1) it generates a set of points that satisfies both geometric and numerical conditions imposed on the physical domain. (2) it builds a robust and well-shaped meshes over this point set, e.g., a triangulation of the point set. Most mesh generation algorithms merge the two functions, and generate the point set implicitly as part of the mesh generation phase. A most useful technique is to generate point set and its discretization by an iterative refinement. We now discuss *hierarchical refinement* and *Delaunay refinement*, two of the most commonly used refinement methods.

19.3.1 Hierarchical Refinement

The hierarchical refinement uses quadtrees in two dimensions and octtrees in three dimensions. The basic philosophy of using quad- and oct-trees in meshes refinements and hierarchical N-body simulation is the same: adaptively refining the domain by selectively and recursively divide boxes enable us to achieve numerical accuracy with close to an optimal discretization. The definition of quad- and oct-tree can be found in Chapter **??**. Figure 19.3 shows a quad-tree.

In quadtree refinement of an input domain, we start with a square box encompassing the domain and then adaptively splitting a box into four boxes recursively until each box small enough with respect to the geometric and numerical requirement. This step is very similar to quad-tree decomposition for N-body simulation. However, in mesh generation, we need to ensure that the mesh is well-shaped. This requirement makes mesh generation different from hierarchical N-body approximation. In mesh generate, we need to generate a set of smooth points. In the context of quad-tree refinement, it means that we need to make the quad-tree *balanced* in the sense that no leaf-box is adjacent to a leaf-box more than twice its side length.

With adaptive hierarchical trees, we can "optimally" approximate any geometric and numerical spacing function. The proof of the optimality can be found in the papers of Bern, Eppstein, and Gilbert for 2D and Mitchell and Vavasis for 3D. Formal discuss of the numerical and geometric spacing function can be found in the point generation paper of Miller, Talmor and Teng.

The following procedure describes the basic steps of hierarchical refinement.

1. Construct the hierarchical tree for the domain so that the leaf boxes approximate the numerical and geometric spacing functions.

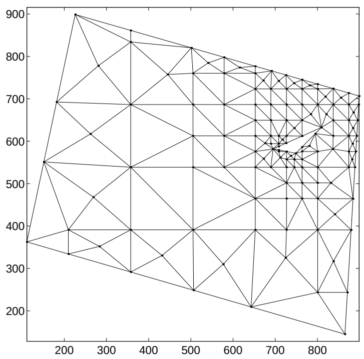


Figure 19.4: A well-shaped mesh generated by quad-tree refinement

2. Balance the hierarchical tree.

3. Warping and triangulation: If a point is too close to a boundary of its leaf box then one of the corners collapses to that point.

19.3.2 Delaunay Triangulation

Suppose $P = \{p_1, \ldots, p_n\}$ is a point set in d dimensions. The convex hull of $d+1$ affinely independent points from P forms a *Delaunay simplex* if the circumscribed ball of the simplex contains no point from P in its interior. The union of all Delaunay simplices forms the *Delaunay diagram*, $DT(P)$. If the set P is not degenerate then the $DT(P)$ is a simplex decomposition of the convex hull of P. We will sometimes refer to the Delaunay simplex as a triangle or a tetrahedron.

Associated with $DT(P)$ is a collection of balls, called *Delaunay balls*, one for each cell in $DT(P)$. The Delaunay ball circumscribes its cell. When points in P are in general position, each Delaunay simplex define a Delaunay ball, its circumscribed ball. By definition, there is no point from P lies in the interior of a Delaunay ball. We denote the set of all Delaunay balls of P by $DB(P)$.

The geometric dual of Delaunay Diagram is the *Voronoi Diagram*, which of consists a set of polyhedra $V_1, \ldots, V_n$, one for each point in P, called the *Voronoi Polyhedra*. Geometrically, V_i is the set of points $p \in \mathbb{R}^d$ whose Euclidean distance to p_i is less than or equal to that of any other point in P. We call p_i the *center* of polyhedra V_i. For more discussion, see [74, 29].

The DT has some very desired properties for mesh generation. For example, among all triangulations of a point set in 2D, the DT maximizes the smallest angle, it contains the nearest-neighbors graph, and the minimal spanning tree. Thus Delaunay triangulation is very useful for computer graphics and mesh generation in two dimensions. Moreover, discrete maximum principles will only exist for Delaunay triangulations. Chew [16] and Ruppert [79] have developed Delaunay refinement algorithms that generate provably good meshes for 2D domains.

Notice that an internal diagonal belongs to the Delaunay triangulation of four points if the sum of the two opposing angles is less than π.

A 2D Delaunay Triangulation can be found by the following simple algorithm: *FLIP algorithm*

- Find any triangulation (can be done in $O(n \lg n)$ time using divide and conquer.)

- For each edge pq, let the two faces the edge is in be prq and psq. Then pq is not an local Delaunay edge if the interior the circumscribed circle of prq contains s. Interestingly, this condition also mean that the interior of the circumscribed circle of psq contains r and the sum of the angles prq and psq is greater than π. We call the condition that the sum of the angles of prq and psq is no more than π *the angle condition*. Then, if pq does not satisfy the angle property, we just flip it: remove edge pq from T, and put in edge rs. Repeat this until all edges satisfy the angle property.

It is not too hard to show that if FLIP terminates, it will output a Delaunay Triangulation. A little addition geometric effort can show that the FLIP procedure above, fortunately, always terminate after at most $O(n^2)$ flips.

The following is an interesting observation of Guibas, Knuth and Sharir. If we choose a random ordering π of from $\{1, ..., n\}$ to $\{1, ..., n\}$ and permute the points based on π: $p_{\pi(1)} \ldots p_{\pi(n)}$. We then incrementally insert the points into the the current triangulation and perform flip if needed. Notice that the initial triangulation is a triangle formed by the first three points. It can be shown that the expected number of flips of the about algorithm is $O(n \log n)$. This gives a randomized $O(n \log n)$ time DT algorithm.

19.3.3 Delaunay Refinement

Even though the Delaunay triangulation maximize the smallest angles. The Delaunay triangulation of most point sets are bad in the sense that one of the triangles is too 'skinny'. In this case, Chew and Ruppert observed that we can refine the Delaunay triangulation to improve its quality. This idea is first proposed by Paul Chew. In 1992, Jim Ruppert gave a quality guaranteed procedure.

- Put a point at the circumcenter of the skinny triangle
- if the circum-center encroaches upon an edge of an input segment, split an edge adding its middle point; otherwise add the circumcenter.
- Update the Delaunay triangulation by FLIPPING.

A point *encroaches* on an edge if the point is contained in the interior of the circle of which the edge is a diameter. We can now define two operations, Split-Triangle and Split-Segment

Split-Triangle(T)

- Add circumcenter C of Triangle T
- Update the Delaunay Triangulation $(P \cup C)$

Split-Segment(S)

- Add midpoint m
- Update the Delaunay Triangulation $(P \cup M)$

The *Delaunay Refinement* algorithm then becomes

- Initialize

  - Perform a Delaunay Triangulation on P
  - IF some segment, l, is not in the Delaunay Triangle of P, THEN Split-Segment (l)

- Repeat the following until $\alpha > 25°$

  - IF T is a skinny triangle, try to Split(T)
  - IF C of T is 'close' to segment S then Split-Seg(S)
  - ELSE Split Triangle (T)

The following theory was then stated, without proof. The proof is first given by Jim Ruppert in his Ph.D thesis from UC. Berkeley.

Theorem 19.3.1 Not only does the Delaunay Refinement produce all triangles so that $MIN\ \alpha > 25°$, the size of the mesh it produces is no more than $C * Optimal(size)$.

VI Selected Topics

Chapter 20

The Pentium Bug

20.1 About These Notes on The Pentium Bug

The purpose of these notes is to summarize the 18.337 class on ♡ February 14, 1995 ♡. It includes a brief history on the discovery of the Pentium bug in §2 followed by the details of the Pentium bug flaw in §3. The material appearing in §3 is intended to be a simplification of the proof which appeared in the Coe/Tang paper "It Takes Six Ones to Reach a Flaw."

20.2 The Discovery of The Bug

The first posting about the Pentium bug was by Professor Nicely at the Lynchburg College of Virginia. It seems that Dr. Nicely was working with twin primes – numbers such as { 11, 13 }, { 29, 31 }, or { 107, 109 }. Specifically, he was going after the sum:

$$S = \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \frac{1}{13} + \frac{1}{17} + \frac{1}{19} + \cdots$$

This sum is mathematically known to be finite. Dr. Nicely had been running his program to compute this sum on several different computers. But when he ran the program on a Pentium machine, he got a different answer than he expected – even though he was using double precision, he was getting worse results than would be expected from single precision numbers. Eventually, Dr. Nicely narrowed the the source of the error to a part of his code that computed the terms in the sum. (i.e. $\frac{1}{p}$) He knew that his code was correct, so on October 10, 1994 he posted his conclusion that the Pentium chip contained a division flaw.

This posting caught the attention of Dr. Timothy Coe at Vitesse, who decided that this was a good opportunity to discover how the chip works. He asked for people to send him examples of the bug, and successfully put together a model of what Intel was doing.

20.3 The Pentium Bug

What Dr. Coe discovered was that Intel was doing division using the Radix-4 version of the Sweeney, Robertson, and Tocher (SRT) algorithm. In order to get an idea of the magnitude of the bug that we will be discussing, consider the following example: The fraction $\frac{4195835}{3145727}$ should evaluate to 1.33382, but a flawed Pentium chip returns 1.33374. This is a relative error of 6.1E-5 or about 61 ppm – roughly twice the concentration of carbon monoxide allowed in ambient air by national ambient air quality standards [1]. It is interesting that some software developers have come up with ways to deal with this annoying bug. For example, there is a patch available for MATLAB which detects at-risk denominators, multiplies the numerator and denominator by $\frac{15}{16}$, and then performs the division. The remainder of this section is devoted to a description of the details of the Radix-4 SRT algorithm and the Pentium Bug.

20.4 Introduction

This is text from my paper. –Alan

20.5 Introduction

Quite properly, the risk to users emerged as the most popular issue in the discussion of the Intel Pentium flaw. Unfortunately, evaluating this risk is probably infeasible. Other popular issues include Intel's public relations policy, the story of how Nicely found the bug, how Coe succeeded in reverse engineering the algorithm based on "bad" numerators and denominators, and those ubiquitous Pentium jokes [14]. Vaughan Pratt [73], holds the record for the most extensive computational experiments and classifications of the bug.

We also wish to emphasize that despite all of the jokes, the bug is far more subtle than many people realize. Andrew Wiles had a "bug" in the first public draft of his proof of Fermat's Last Theorem – his work was too important for anybody to laugh. The bug in the Pentium was an easy mistake to make, and a difficult one to catch. One way to catch it directly is Kahan's [49] SRT division tester. Kahan's tester chooses arguments more cleverly than random testing, increasing the likelihood of discovering whether an algorithm is somehow broken. His tester concentrates on the fenceposts, which is always a good place to look for difficulties. Randy Bryant [13] has a different sort of tester. He uses binary decision diagrams as a check of the validity of a PD table. This is the table in which the bug appears. His algorithm explicitly checks that partial remainders remain within the critical region. It is not immediately clear that this tester could be directly used on the Pentium chip.

Sometimes a mistake is itself uninteresting other than that it causes trouble and that it needs to be fixed. For example, a wrong turn while driving in a familiar neighborhood can waste time. On occasion, a mistake is itself inherently interesting. Proceeding with our example, a wrong turn in an unfamiliar neighborhood may lead to interesting discoveries. With all of the media coverage, one message that hardly anyone has noticed is that the bug itself is mathematically interesting. What a shame that, to the best of my knowledge, at the time of writing this paper, there are only a handful of mathematicians that actually understand the bug.

Suppose the Pentium chip had never built, but a mathematician had a complete specification of the algorithm including the flaw. Could this mathematician without the aid of a computer readily discover flawed numerators and denominators? My opinion is that the world's leading mathematicians might find them, but I am not so sure. They are difficult to find using traditional non-computer oriented mathematics.

This article is a self-contained mathematical discussion of the bug, and only the bug. Sections 2 through 4 contain a complete mathematical specification of the algorithm which would be sufficient for the reader to try to see if he can devise buggy numerators and denominators without ever touching a Pentium chip.

[1] National Primary and Secondary Ambient-Air-Quality Standard, 1983

We begin with a quick discussion of radix (or base) 4 SRT division, and its carry-save implementation. A very nice tutorial on the subject was recently written by David Goldberg [?]. We then give a simplified proof of the Coe/Tang [21] result that it takes six ones to reach a flaw. The first part of the proof begins with a simple analysis of the inequalities that either prevent you or allow you access to an erroneous table entry. The second part of the proof is an arithmetic puzzle not so very different in spirit from the sort found in recreational mathematics, where one has to replace letters with digits to make a correct sum as in this one from a collection by James Fixx [32]

```
   S E N D
 + M O R E
 ---------
 M O N E Y
```

We conclude with an explanation of why the Pentium is always guaranteed to have an absolute error that is bounded by `5e-5`. when the inputs are in the standard interval $[1, 2)$.

20.6 SRT Division

We now present the radix 4 version of the Sweeney, Robertson, and Tocher (SRT) division algorithm. This algorithm computes a radix 4 quotient where the digits are not the set $\{0, 1, 2, 3\}$ as might be expected from base 4, but rather $\{-2, -1, 0, 1, 2\}$. The extra digit introduces a very useful "slack" into the computation, For multiplication, the digits 0, ± 1, and ± 2 are also computationally more convenient than the digit 3 would be, say.

Letting p and d denote the numerator and the denominator, we may as well assume that $1 \leq p, d < 2$. The SRT division algorithm may be expressed succinctly in conceptual pseudocode:

Radix 4 SRT Division

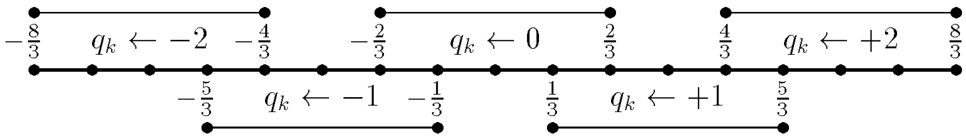
$p_0 := p$
for $k = 0, 1, \ldots$
  "Lookup" a digit $q_k \in \{-2, -1, 0, 1, 2\}$ in such a way that
    $p_{k+1} := 4(p_k - q_k d)$
  satisfies $|p_{k+1}| \leq \frac{8}{3}d$
end
$p/d =_{i=0}^{\infty} q_i/4^i$.

How the lookup table works on the Pentium is not important yet. It is very clever, but the discussion is postponed to Section 20.8. We will remark, that on the Pentium chip, by an accident in the table lookup, it is possible to obtain a q_k which fails to produce a p_{k+1} with absolute value $\leq \frac{8}{3}d$. It is a popular misconception that SRT can recover from such an accident.

We note that if we replace the set $\{-2, -1, 0, 1, 2\}$ with $\{0, \ldots, 9\}$, and the 4 with a 10, and finally the inequality $|p_{k+1}| \leq \frac{8}{3}d$ with $p_{k+1} \in [0, 10d)$, then we derive ordinary base 10 long division.

20.7 Understanding why SRT works

It is easy to see that if $|p_k| \leq \frac{8}{3}d$ then at least one of the five values of $p_k - q_k d$ has absolute value $\leq \frac{2}{3}d$. The figure below (with unit length d) shows this clearly. The indicated q_k value translates each interval to the $q_k \leftarrow 0$ interval. Overlap between upper and lower intervals indicates that an arbitrary choice may be made. The $q_k \leftarrow 0$ interval, when scaled up by a factor of 4, remains within the interval $|p| \leq \frac{8}{3}d$.



We therefore may run the algorithm ad infinitum, i.e. we can find a q_k for which

$$p_{k+1} = 4(p_k - q_k d) \tag{20.1}$$

has absolute value $\leq \frac{8}{3}d$.

Since equation (20.1) is equivalent to

$$\frac{p_k}{d}4^{-k} = \frac{q_k}{4^k} + \frac{p_{k+1}}{d}4^{-(k+1)},$$

we may prove by induction that

$$\frac{p}{d} = \left(q_0 + \frac{q_1}{4} + \cdots + \frac{q_{k-1}}{4^{k-1}}\right) + \frac{p_k}{d}4^{-k}. \tag{20.2}$$

Letting $k \to \infty$ in Equation (20.2) proves that the SRT algorithm computes the correct quotient. $\square$

There is a popular misconception that the SRT algorithm is capable of correcting for "mistakes" by using the redundant set of digits. Though the overlapping in the regions may allow for two choices of digits, if an invalid digit is chosen, the algorithm can never recover.

A consequence of Equation (20.2) is that

$$p_k = d^{\infty}_{i=k} q_i/4^{i-k} = d(q_k + \frac{q_{k+1}}{4} + \frac{q_{k+2}}{4^2} + \ldots).$$

Summing the geometric progression with the extreme choices of all $q_i = +2$ or all $q_i = -2$ shows that the requirement $|p_k| \leq \frac{8}{3}d$, is not an arbitrary choice, but a necessary ingredient in the algorithm. On the Pentium, for certain values of d and for certain values of p_k just a little smaller than $\frac{8}{3}d$, the chip supplies the value 0 for q_k rather than 2. The value of p_{k+1} would then be nearly $10d$ which is way outside of the range representable by our geometric progression. A consequence is that when the Pentium looks up 0 instead of 2, there is no way to recover.

A small observation that we will use later is

Lemma 20.7.1 *If $q_k = 2$, then $p_{k+1} \leq p_k$.*

Proof This is a simple consequence of the bound $p_k \leq \frac{8}{3}d$. $\square$

20.8 Quotient digit selection on the Pentium

The Pentium uses a lookup table to obtain q_k. Rather than using p_k and d directly in the lookup table, it uses approximations P_k and D to p_k and d respectively. P_k is a binary number of the form `xxxx.yyy`, i.e. an integer multiple of 1/8, while D is a binary number of the form `x.yyyy`, i.e. an integer multiple of 1/16. The number P_k is obtained from the carry-save representation to be explained momentarily. We shall see that we can guarantee that

$$P_k \le p_k < P_k + \frac{1}{4} \text{ and } D \le d < D_+ \equiv D + \frac{1}{16}. \tag{20.3}$$

Notice that D and D_+ are defined by the second inequality, but the first inequality allows for two possible values of P_k given p_k. The choice that is made is defined in Equation 20.5.

The lookup table [21, 85] that computes q_k as a function of P_k and D appears in Figure 1 below. The five colors indicate the five digits $+2, +1, 0, -1, -2$ from top to bottom. The five "explosion symbols" at the top of the table indicate table entries that must be 2 but erroneously return 0 on flawed Pentium chips. Note the five columns are completed using the Coe model, while other columns have ambiguous entries. The five entries on the bottom of the table with white borders can not be reached. Any proposed verifier of the PD table must be cognizant of the fact that these entries are unreachable. As Pratt writes [73]

> One thinks of testing as being as good as verification *if* one could test all possible cases. As a weakened version of this, a comprehensive test should exercise every device and/or line of code in the system.
>
> The Pentium bug reveals a serious limitation of this approach. There is of course no data that can exercise unreachable code or table entries. Thus if one believes that the five "missing" entries are unreachable, then no attempt will be made to produce a test for this case. Hence missing entries are likely to be overlooked by any *fabricated* set of test cases.

The operative word is the word *believes* in the quotation above.

The five bad entries occur for those divisors d for which D has any of the five values 17/16, 20/16, 23/16, 26/16, 29/16. These are the values which allow us a peek at how the Pentium chip is performing division. For the other D values, there is no algorithm pathology, so there is no independent way to be sure which digits are selected in the overlap regions. Hence the white boxes in the interior of the lookup table indicating that two possible choices are equally correct.

For the remainder of this paper, we are only concerned with the five bad columns in the lookup table. There are the columns with the explosion symbols in Figure 1. Mathematically, they are the columns where $\frac{2}{3}D_+$ is an integer multiple of 1/8. Following Coe and Tang [21], a flawed entry exists in the box corresponding to the value

$$\text{buggy entry} = P_{\text{Bad}} = \frac{8}{3}D_+ - \frac{1}{8}.$$

The information expressed in colors in Figure 1 may be expressed succinctly with thresholds by identifying which of five intervals P_k falls in:

$$P_q^{\text{Min}} \le P_k \le P_q^{\text{Max}}. \tag{20.4}$$

The thresholds according to [21] are

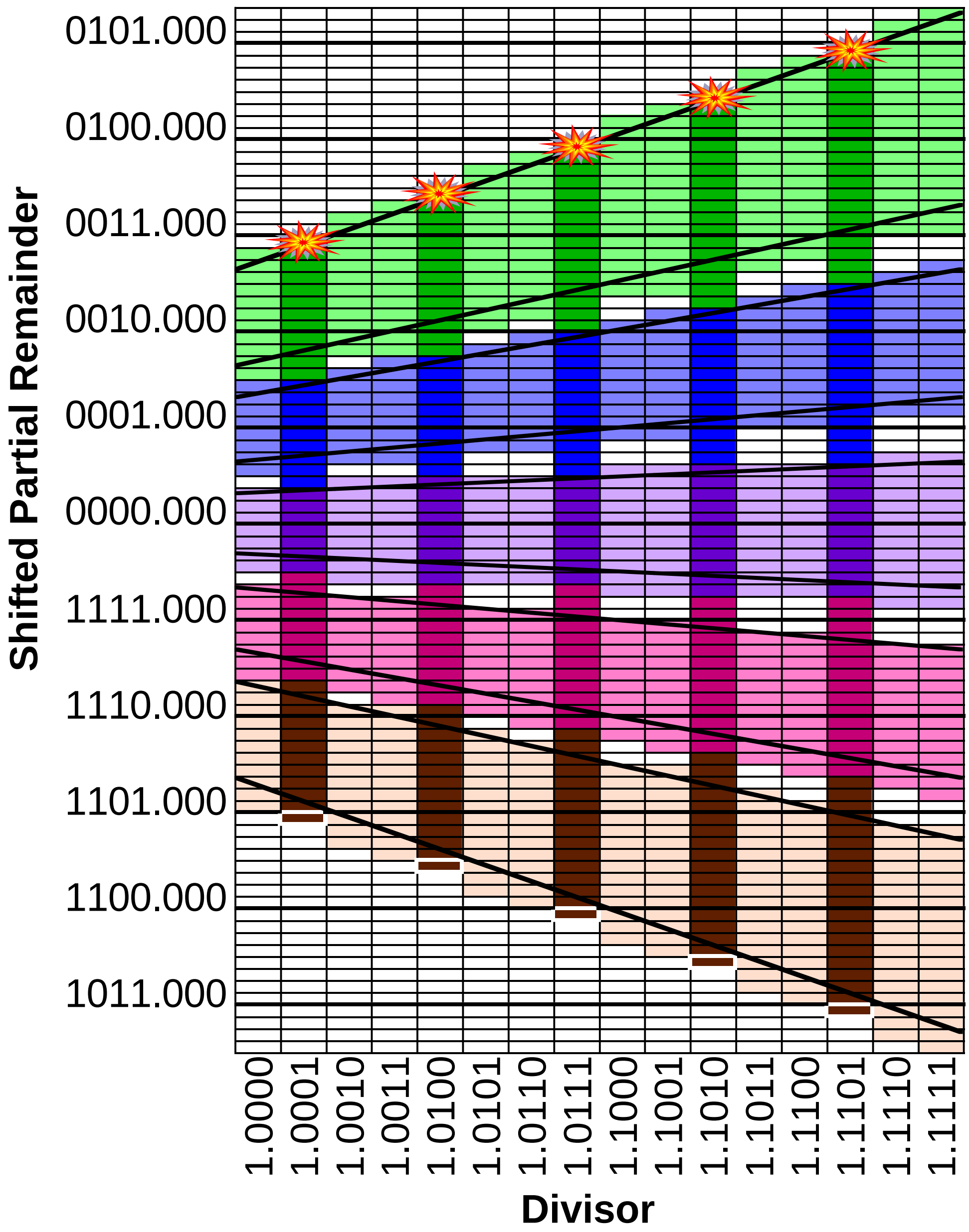


Figure 20.1: The quotient digit $q_k \in \{-2, -1, 0, 1, 2\}$ is a function of P_k and D. During one SRT division, D is fixed, so entries are accessed within one column. The five trapezoidal bands (green, blue, etc.) indicate the digits from $+2$ to -2 in decreasing order from the top. The five flawed entries that should be $+2$ but which instead are accidentally 0 are indicated by the explosion symbols at the top of the five columns. In these columns we know from the Tim Coe model the values of the entries. In other columns, there are empty squares in the interior where two possible entries are valid. Towards the top and bottom of the table are entries that a correctly working Pentium can not access. Note the five entries towards the bottom without borders. They also are not accessed.

q	P_q^{Min}	P_q^{Max}
-2	$-\frac{8}{3}D_+$	$-\frac{1}{4}-\frac{4}{3}D_+$
-1	$-\frac{1}{8}-\frac{4}{3}D_+$	$\lfloor -\frac{1}{4}-\frac{1}{3}D_+ \rfloor$
0	$\lfloor -\frac{1}{8}-\frac{1}{3}D_+ \rfloor$	$\lceil -\frac{1}{8}+\frac{1}{3}D_+ \rceil$
1	$\lceil \frac{1}{3}D_+ \rceil$	$-\frac{1}{8}+\frac{4}{3}D_+$
2	$\frac{4}{3}D_+$	$-\frac{1}{8}+\frac{8}{3}D_+$

,

where the floor and ceiling symbols round to multiples of 1/8.

It is easy to check that the q chosen in this manner satisfies the constraints of the algorithm specified in Section 2. We stress once again that the thresholds given in the table above are meant to apply only in the five buggy columns.

The computation of P_k: Carry-Save addition

Imagine adding 100 numbers on a calculator. On many calculators after typing $x_1 + x_2$ the sum would be displayed, then folding in x_3 the new sum is obtained, etc. On computers it is convenient to avoid the carry propagation by leaving the result in so-called "carry-save" format (See [23, pp.668–669]. In this format $x_1 + x_2$ is represented as $s_2 + c_2$. When we add in x_3 the result is represented as $s_3 + c_3$, etc. The s_i and c_i are known as the sum and carry word. The basic idea is that when computing the sum of $s_2 + c_2 + x_3$ in binary, every column can add up to $0, 1, 2$, or 3 so the modulo 2 sum of the result (0 or 1) is stored in the sum word, and the carry bit is stored in the carry word.

Here is an example:

$$
\begin{array}{c|ccccc}
x_1 & 0 & 1 & 0 & 1 & 1 \\
x_2 & 0 & 1 & 1 & 0 & 1 \\
\hline
s_2 & 0 & 0 & 1 & 1 & 0 \\
c_2 & 1 & 0 & 0 & 1 & 0 \\
x_3 & 0 & 1 & 1 & 1 & 0 \\
\hline
s_3 & 1 & 1 & 0 & 1 & 0 \\
c_3 & 0 & 1 & 1 & 0 & 0
\end{array}
$$

Generally speaking `sum(a,b,c)` is $a + b + c \bmod 2$ in mathematical language, and $a \oplus b \oplus c$ in a more computer science style language. `carry(a,b,c)` may be expressed as $a + b + c \geq 2$ in a matlab sort of language, or as $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$. The numbers c_k and s_k constitute the carry-save representation of ${}_{i=1}^{k} x_k$. This representation is not unique.

On the Pentium, p_k is represented in the carry-save form $c_k + s_k$ so that the expression $p_{k+1} = p_k - q_k d$ is computed using a carry-save adder. If q_k is positive $-q_k d$ is represented as the one's complement of $q_k D$.[2]

In general $p_k - q_k d$ is obtained from d by a combination of shifting and/or ones complementing or zeroing. Since these are fast operations on a computer, this explains why the digit set $\{-2, -1, 0, 1, 2\}$ is so useful. To perform addition correctly, if a one's complement number is used, a 1 is added in to the least significant carry bit before shifting. We will see this in the example.

Given that $p_k = c_k + s_k$, it is natural to define

$$P_k \equiv C_k + S_k, \tag{20.5}$$

where C_k and S_k represent c_k and s_k respectively rounded down to the nearest bit. Since

$$C_k \leq c_k < C_k + \frac{1}{8} \text{ and } S_k \leq s_k < S_k + \frac{1}{8},$$

[2] One's complement represents integers by complementing every bit. Since this is not a "true" negative, a correction term must be added at some point in the calculation.

we can conclude that $P_k \le p_k + \frac{1}{4}$.

RADIX 4 SRT DIVISION WITH CARRY-SAVE ADDITION PENTIUM STYLE

$D := \lfloor d \rfloor_{1/16}$
$S_0 := p_0, \quad C_0 := 0$
for $k = 0, 1, \ldots$
  $P_k := \lfloor c_k \rfloor_{1/8} + \lfloor s_k \rfloor_{1/8}$
  $q_k :=$ table-lookup(P_k, D)
  compute $(-q_k d)$ by zeroing, shifting, and/or ones complementing
  $c_{k+1} + s_{k+1} := 4(c_k + s_k + (-q_k d))$ carry-save addition and shift
    Correct for one's complement in c_{k+1} if $q_k < 0$
end
$p/d =_{i=0}^{\infty} q_i/4^i$.

The following example illustrates the division of 1.875 by 1.000 using fewer bits than is actually used in the Pentium. (The actual number of bits used in the Pentium is important (see [73, Section 2, last paragraph]), but not for our concerns here.) Of course, dividing by 1 is trivial, but the important features of the algorithm are illustrated below.

1.875=	`0001.111 00000000000`	s_0=1.875 in binary
	`0000.000 00000000000`	c_0 is initially set to 0
-2x1=	`1101.111 11111111111`	2 (from lookup table, where $P_0 = C_0 + S_0$) in one's complement
	`0000.011 11111111100`	s_1 is the sum from above with the two place shift
	`1111.000 00000000100`	c_1 is the carry, with shift, and one's complement correction bit
-(-1)x1=	`0001.000 00000000000`	$S_1 = .011 = 3/16$, $C_1 = 1111.000 = -1$, $P_1 = -13/16$.
	`1001.111 11111100000`	
	`1000.000 00000100000`	no correction bit since no one's complement in previous iteration
-2x1 =	`1101.111 11111111111`	
	`0000.000 00011111100`	
	`1111.111 11100000100`	
-0x1=	`0000.000 00000000000`	
	`1111.111 11111100000`	
	`0000.000 00000100000`	
	`0000.000 00000000000`	
	`1111.111 11100000000`	
	`0000.000 00100000000`	

We have thus computed the representation $1.875 = 2 - \frac{1}{4} + \frac{2}{16} + \frac{0}{32} + \ldots$.

20.9 Analyzing the bug

20.9.1 It is not easy to reach the buggy entry

We proceed to prove a lemma which states that the buggy entry P_{Bad} can only be reached from the entry below it in the PD table. This entry just below the bad entry plays the role of a foothold. The existence of this foothold is a subtle phenomenon that may not have been readily guessed from general properties of SRT division. I believe that its existence has surprised everybody. Any thought that each table entry is somehow equally likely to be reached is very wrong.

The table entries that play an important role in reaching the bug are

Description	Symbol	Value
Buggy Entry	P_{Bad}	$-\frac{1}{8}+\frac{8}{3}D_+$
Foothold	$P_{\text{Bad}}-\frac{1}{8}$	$-\frac{1}{4}+\frac{8}{3}D_+$
Top of $q=-2$ region	P_{-2}^{Max}	$-\frac{1}{4}-\frac{4}{3}D_+$
Top of $q=-1$ region	P_{-1}^{Max}	$-\frac{1}{4}-\frac{1}{3}D_+$

The result that we proceed to prove in Lemma 20.9.1 below is that one can only reach the buggy entry from the foothold. We will also show that it is possible to reach the foothold in four ways: from the top of the $q=-2$ region, the top of the $q=-1$ region, from the foothold itself or the entry below the foothold. However, if the foothold is reached from the latter two of the four routes, the buggy entry can not be reached on the next step. Therefore, there are only two viable routes to reach the bug. It is worth emphasizing that reaching the foothold is necessary for reaching the buggy entry, but not sufficient. It is quite possible to reach the foothold, but not hit the buggy entry on the next step.

We now proceed with our analysis. Directly from the definition of D_+ given in Equation 20.3, we can write the following identities for $q_k D_+$ in terms of the binary expansion of $d = 1.d_1d_2d_3d_4d_5\ldots$:

$$\begin{array}{rcll} 2D_+ &=& 0\,0\,1\,d_1.\,d_2\,d_3\,d_4\;0 & +\frac{1}{8} \\ D_+ &=& 0\,0\,0\;1.\;d_1\,d_2\,d_3\,d_4 & +\frac{1}{16} \\ 0D_+ &=& 0\,0\,0\;0.\;\;0\;0\;0\;0 & , \\ -D_+ &=& 1\,1\,1\;0.\;\bar{d}_1\,\bar{d}_2\,\bar{d}_3\,\bar{d}_4 & \\ -2D_+ &=& 1\,1\,0\,\bar{d}_1.\,\bar{d}_2\,\bar{d}_3\,\bar{d}_4\;0 & \end{array} \tag{20.6}$$

where the negatives numbers are expressed in two's complement notation.[3] The chopping is a round down process, but D_+ is akin to a round up, hence the existence of the $\frac{1}{8}$ and $\frac{1}{16}$ terms.

Therefore in terms of P_k and D_+, we have the version of Equation (20.1) that is actually computed on the Pentium:

$$P_{k+1} = 4(P_k - q_k D_+) + R_k, \tag{20.7}$$

where R_k is determined by a few higher order bits. Working out the exact value of R_k requires careful attention to the algorithmic details. The reader may verify that

$$R_k = 0.0s_4s_5 + 0.0c_4c_5 + \left\{\begin{array}{l} 0.0\;d_5\,d_6 \\ 0.0\;0\;\;d_5 \\ 0 \\ 0.0\;0\;\;\bar{d}_5 \\ 0.0\;\bar{d}_5\,\bar{d}_6 \end{array}\right\} + \frac{1}{8}\times\text{carry-over}(s_6, c_6, \left\{\begin{array}{c} d_7 \\ d_6 \\ 0 \\ \bar{d}_6 \\ \bar{d}_7 \end{array}\right\}) + \left\{\begin{array}{c} -1/2 \\ -1/4 \\ 0 \\ 0 \\ 0 \end{array}\right\},\ \text{if} \left\{\begin{array}{c} q=-2 \\ q=-1 \\ q=0 \\ q=1 \\ q=2 \end{array}\right.$$

where the s_i, c_i, and d_i are the appropriate bits in the sum, carry, and divisor respectively.[4] The function carry-over(b_1, b_2, b_3) is 1 if at least two of its arguments are 1. The last correction term of $-1/2$ and $-1/4$ is a direct consequence of Equation (20.6).

[3]Two's complement represents integers by reducing modulo a power of 2. Add D_+ and $-D_+$ or add $2D_+$ and $-2D_+$ (and multiply by 16 to remove the binary point) to see that the sum is a power of 2.

[4]We regret the abuse of notation – everywhere else in this paper s_i refers to all of the bits in the sum word of the ith iteration, but here it represents the ith bit during an unspecified iteration.

Taking the maximum value of the parameters we see that

$$R_k \leq R_k^{\text{Max}} \equiv \left\{ \begin{array}{c} 3/4 \\ 3/4 \\ 7/8 \\ 1 \\ 5/4 \end{array} \right\}, \quad \text{if} \quad \left\{ \begin{array}{c} q=-2 \\ q=-1 \\ q=0 \\ q=1 \\ q=2 \end{array} \right. . \tag{20.8}$$

We assume that D_+ is a multiple of 3/16 i.e., it corresponds to one of the five flawed table entries. For each value of q_k, we may tabulate the largest possible P_{k+1} that may be reached as a function of the extreme P_k by using equations (20.7) and (20.8). Therefore the table below is a transition table that shows how high one can reach in the table given the previous value of q_k. This is our first glimpse at just how difficult it is to reach the bad entry P_{Bad}.

	If P_k is at most $\downarrow$,	then P_{k+1} is at most $\downarrow$	can reach the foothold
q_k	P_k	$P_{k+1}^{\text{Max}} = 4(P_k - q_k D_+) + R_k^{\text{Max}}$	
-2	$P_{-2}^{\text{Max}} = -\frac{4}{3}D_+ - \frac{1}{4}$	$\frac{8}{3}D_+ - \frac{1}{4} = P_{\text{Bad}} - \frac{1}{8}$	yes
-1	$P_{-1}^{\text{Max}} \leq -\frac{1}{3}D_+ - \frac{1}{4}$	$\frac{8}{3}D_+ - \frac{1}{4} = P_{\text{Bad}} - \frac{1}{8}$	yes
0	$P_0^{\text{Max}} \leq \frac{1}{3}D_+$	$\frac{4}{3}D_+ + \frac{7}{8} < P_{\text{Bad}} - \frac{1}{8}$	no
1	$P_1^{\text{Max}} = \frac{4}{3}D_+ - \frac{1}{8}$	$\frac{4}{3}D_+ + \frac{1}{2} < P_{\text{Bad}} - \frac{1}{8}$	no
	If P_k is at most $\downarrow$,	then P_{k+1} is at most $\downarrow$	reach buggy entry
2	$P_{\text{Bad}} - \frac{1}{8} = \frac{8}{3}D_+ - \frac{1}{4}$	$\frac{8}{3}D_+ + \frac{1}{4} > P_{\text{Bad}}$	yes
2	$P_{\text{Bad}} - \frac{1}{4} = \frac{8}{3}D_+ - \frac{3}{8}$	$\frac{8}{3}D_+ - \frac{1}{4} = P_{\text{Bad}} - \frac{1}{8}$	no

(20.9)

The two "less than" inequalities in the table above are a simple consequence of $D_+ > 1$. In terms of Figure 1, the above table states that it is impossible to reach the explosion symbol even if you are at the very top of the brown, pink, purple, or blue regions. The only way to reach the explosion symbol is from the green entry immediately below it. Therefore, we denote this entry the foothold.

Lemma 20.9.1 *The sequence of P's and corresponding q's that lead to the bad entry are given below:*

$$\begin{array}{lccccc} P\!: & P_{-2}^{Max}/P_{-1}^{Max} & \longrightarrow & \left\{P_{Bad} - \frac{1}{8}\right\}_{m\geq 1} & \longrightarrow & P_{Bad} \\ q: & -2/-1 & \longrightarrow & \{2\}_{m\geq 1} & \longrightarrow & \text{(bad entry)} \cdot \\ R: & & R = R^{\max} & & R = R^{\max} - \frac{3}{8} & \end{array} \tag{20.10}$$

Here the subscript m in the middle column denotes a repetition of the entry $m \geq 1$ times, and the first column indicates that either a P_{-2}^{Max} or P_{-1}^{Max} may start the path to the bug corresponding to either $q = -2$ or $q = -1$ respectively.

Proof Our proof goes from right to left. The table in (20.9) shows that we can only reach P_{Bad} from $P_{\text{Bad}} - \frac{1}{8}$. The table also shows that for $q = -2$ or $q = -1$, we can just barely reach $P_{\text{Bad}} - \frac{1}{8}$, but only when $P_k = P_k^{\text{Max}}$ and $R_k = R_k^{\text{Max}}$. $P_{\text{Bad}} - \frac{1}{8}$ may also be reached from $P_{\text{Bad}} - \frac{1}{4}$, but if $P_k = P_{\text{Bad}} - \frac{1}{4}$ then $P_{\text{Bad}} - \frac{1}{4} \leq p_k < P_{\text{Bad}}$, and Lemma 20.7.1 guarantees that while the sequence produces 2's all future p_k will also satisfy $p_k < P_{\text{Bad}}$ which precludes reaching the flawed entry, i.e., reaching k with $P_k = P_{\text{Bad}}$. □

20.10 The "Send More Money" Puzzle for the Pentium

Theorem 20.10.1 (Coe,Tang) *A flawed table entry can only be reached if the divisor $d = d_1d_2d_3\ldots$ has the property that the six consecutive bits from d_5 to d_{10} are all ones. (Of course $1.d_1d_2d_3d_4$ must be one of the five bad values.)*

Proof

First assume that $m = 1$. (We later show in Lemma 20.10.1 that this is the only possibility.) The table below illustrates the carry-save division calculation that leads to the buggy entry. We devised a subscript notation to facilitate following the progression of fill-in. Key bits are subscripted with a spade ($\spadesuit$) symbol, as in $\alpha_\spadesuit$. Then in sequence, we let $\alpha_1, \alpha_2, \ldots$ denote bits we can fill in more or less mechanically based on the carry-save division process. The next key bit is subscripted $\beta_\spadesuit$ and then we fill in $\beta_1, \beta_2, \ldots$ mechanically in sequence. The only ideas used to fill in the mechanical entries are the shifted carry-sum division process and the possible shifting or complementing of the bits in d. Asterisks ($*$) or blank spaces indicate entries whose values are not of interest. Also of no interest are values to the left of the binary point.

Case I: Reaching the bug from -2

	s_{j-2}	$.*$	$*$	$*$	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}
$\boxed{q=-2}$	c_{j-2}	$.*$	$*$	$*$	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}
	$2d$	$.d_2$	d_3	d_4	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}
	s_{j-1}	$.*$	1_{α_1}	1_{α_1}	$1_{\gamma_\spadesuit}$	1_{δ_1}	1_{ϵ_1}		
$\boxed{q=2}$	c_{j-1}	$.1_{\alpha_1}$	1_{α_1}	$1_{\beta_\spadesuit}$	$1_{\gamma_\spadesuit}$	1_{δ_1}	1_{ϵ_1}		
	$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0_{α_1}	0_{α_1}	0_{γ_2}		
	s_j	$.*$	0_{γ_1}	0_{δ_2}					
$\boxed{\text{bug}}$	c_j	$.*$	$1_{\delta_\spadesuit}$	$1_{\epsilon_\spadesuit}$					

Case II: Reaching the bug from -1

s_{j-2}	$.*$	$*$	$*$	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}	
c_{j-2}	$.*$	$*$	$*$	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}	$\boxed{q=-1}$
d	$.d_1$	d_2	d_3	$1_{\alpha_\spadesuit}$	$1_{\alpha_\spadesuit}$	1_{γ_1}	1_{δ_2}	1_{ϵ_2}	
s_{j-1}	$.*$	1_{α_1}	1_{α_1}	$1_{\gamma_\spadesuit}$	1_{δ_1}	1_{ϵ_1}			
c_{j-1}	$.1_{\alpha_1}$	1_{α_1}	$1_{\beta_\spadesuit}$	$1_{\gamma_\spadesuit}$	1_{δ_1}	1_{ϵ_1}			$\boxed{q=2}$
$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0_{α_1}	0_{γ_2}	0_{δ_3}			
s_j	$.*$	0_{γ_1}	0_{δ_2}						
c_j	$.*$	$1_{\delta_\spadesuit}$	$1_{\epsilon_\spadesuit}$						$\boxed{\text{bug}}$

Explanation:

- $\alpha_\spadesuit$ Since R (Lemma 20.9.1) must equal R^{max} we must have c_4, c_5, s_4, s_5, and the two d bits equal to 1.
- $\beta_\spadesuit$ $8(P_{\text{Bad}} - \frac{1}{8})$ is even. (It can be 22, 26, 30, 34, or 38.) To obtain an even number, we must add a 1 to the 1_{α_1} immediately above.
- $\gamma_\spadesuit$ $R = R^{\max} - \frac{3}{8}$ (Lemma 20.9.1). The 0_{α_1} immediately below reduces R by 2/8, and a 0 in the $\gamma_\spadesuit$ position would reduce R by another 2/8, giving 4/8.
- $\delta_\spadesuit$ $8P_{\text{Bad}} \in \{23, 27, 31, 35, 39\}$ is 3 (mod 4).
- $\epsilon_\spadesuit$ $8P_{\text{Bad}} \in \{23, 27, 31, 35, 39\}$ is 3 (mod 4).

The conclusion that we have so far is that if we begin with $q = -2$, we must then have that d_5, d_6, d_7, d_8, d_9 are all 1. If we begin with $q = -1$, then we conclude that d_5, d_6, d_7, d_8 are all 1.

We are almost, but not quite finished. Having all those ones at the $q = -2$ or $q = -1$ step allows us to go back yet another step. A quick analysis shows that the line above s_{j-2} can only be d or $2d$ and that it too has many ones. We can then guarantee that d_9 and d_{10} are also 1. □

Lemma 20.10.1 *The value for m in Lemma 20.9.1 must be one.*

Proof We proceed with the same sort of send-more-money puzzle. The assumption of more than one $q = 2$ in sequence yields a contradiction. We encourage readers to try to find the contradiction themselves. It is quite a puzzle. In the diagram below we do not use subscripts for values that we have already obtained from α, β, and γ in Theorem 20.10.1. We then continue the numbering convention with ζ.

Case I: Not reaching the bug from -2, 2, 2
Sequence leading to contradiction

	s_{j-2}	$.*$	$*$	$*$	1	1	1	1_{ζ_5}
$\boxed{\text{q=−2}}$	c_{j-2}	$.*$	$*$	$*$	1	1	1	1_{ζ_5}
	$2d$	$.d_2$	d_3	d_4	1	1	1	1_{ζ_5}
	s_{j-1}	$.*$	1	1	1	1_{ζ_4}	$*$	1_{ζ_7}
$\boxed{q=2}$	c_{j-1}	$.1$	1	1	1	1_{ζ_4}	$*$	1_{ζ_7}
	$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0	0	0	0_{ζ_6}
	s_j	$.*$	0	0_{ζ_2}	$1_{\zeta_\spadesuit}$	0_{ζ_7}		
$\boxed{\text{q=2}}$	c_j	$.*$	1_{ζ_3}	0_{ζ_1}	$1_{\zeta_\spadesuit}$	$1_{\eta_\spadesuit}$		
	$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0	0	0	
	s_{j+1}			1_{η_1}	$0_{\theta_\spadesuit}$			
	c_{j+1}			1_{η_2}				

Case II: Not reaching the bug from -1, 2, 2
Sequence leading to branch

s_{j-2}	$.*$	$*$	$*$	1	1	1	
c_{j-2}	$.*$	$*$	$*$	1	1	1	$\boxed{\text{q=−1}}$
d	$.d_1$	d_2	d_3	1	1	1	
s_{j-1}	$.*$	1	1	1			
c_{j-1}	$.1$	1	1	1			$\boxed{\text{q=2}}$
$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0	0		
s_j	$.*$	0	$\spadesuit$	1			
c_j	$.*$	$\spadesuit$	$\spadesuit$	1			$\boxed{\text{q=2}}$
$-2d$	$.\bar{d}_2$	$\bar{d}_3$	$\bar{d}_4$	0	0		

Explanation (Case I):

$\zeta_\spadesuit$ If the next entry is the bug, $R = R^{max} - \frac{3}{8}$, otherwise $R = R^{max} - \frac{4}{8}$. Either way we need these two ones or we lose $\frac{5}{8}$.

$\eta_\spadesuit$ We have already lost $\frac{4}{8}$ so the next digit is a 2 and we can not lose any more.

$\theta_\spadesuit$ This is the contradiction! It must be a 0 because, if it were 1, then we would have a 1 in the line above where we already have a 0. On the other hand, to reach either the foothold or the buggy entry on the next step, the value must be 1 or R is too small. This is the contradiction.

Explanation (Case II):
Case II has only been started above. The only two possibilities for the three spades pattern in the diagram above are

$$\begin{matrix} 0 & \spadesuit \\ \spadesuit & \spadesuit \end{matrix} = \begin{matrix} 0 & 1 \\ 0 & 1 \end{matrix} \text{ and } \begin{matrix} 0 & 0 \\ 1 & 0 \end{matrix},$$

since the foothold must be 2 modulo 4. We omit the details here, but following through on the first possibility it bomes clear that the pattern

$$\begin{array}{l} \hline .*\ *\ 1\ 1\ 0 \\ .*\ 0\ 1 \\ .*\ *\ *\ 0\ 0\ 1\ 0 \\ \hline \end{array}$$

perpetuates ad infinitum. Following the bug backwards in the second possibility would show that the pattern of bits for every step when $q = 2$ from the jth iteration until the bug is hit would have the pattern

$$\begin{array}{l} \hline .*\ 0\ 0\ 1\ 1\quad 1 \\ .*\ 1\ 0\ 1\ 1\quad 1 \\ .*\ *\ *\ 0\ 0\ 0\ 1 \\ \hline \end{array},$$

which quickly leads to a contradiction. $\square$

20.11 At least nine steps to failure

Naively, one might have thought that the bad table entries would be hit uniformly at random. If this had been the case, the bug surely have been noticed earlier and would have had the potential to cause more significant damage every instance it went unnoticed.

Even in the worst case of the error, the absolute error in the quotient is roughly $0.00005 = 5e-5$ when dividing two numbers in the interval $1 \leq x < 2$. This is the worst case, the actual errors can be far smaller. Vaughan Pratt [72] has exhaustively computed all the single precision errors and has found that the quotient 14909255/11009918 has an absolute error that is roughly $5e-5$. (In fact it is $4.65e-5$.) The highest relative error that he reported is roughly $6e-5$.

The reason the error is bounded is that, no matter what, the Pentium is guaranteed to compute q_0 through q_7 correctly. This is what we will now prove. The result is a straightforward consequence of the results in the previous section.

In particular, if the bug occurs at step 8, then we saw in Section 6 that step 6 must have the form shown in the diagram below which shows the sum, carry, and $q_k d$ bits respectively starting with bit number 4. Taking into account that all the carry bits are 0 at step 1, it is fairly easy to show (following and up and down "wave" motion) that the bit patterns must fall as in the diagram below. However, since q_1 is positive, it must follows that all of $q_2, \ldots, q_6$ are positive by observing the overlap in the $q_k d$ row from one step to the next. However, we known that $q_6 < 0$ so we have a contradiction. Therefore the pattern shown below as Step 6, which must occur to trigger the bug, can not appear before the seventh step, and hence the bug appears no earlier than the ninth step.

KEY TO DIAGRAM BELOW
START WITH $\spadesuit_0$ AND FOLLOW $\spadesuit_1$ UP THE CHART. THEN START WITH $\heartsuit_0$ AND FOLLOW THE $\heartsuit$S DOWN THE CHART. NEXT FOLLOW THE $\clubsuit$, THE $\diamondsuit$, THEN THE α AND THE β. THIS NOTATION MAKES IT EASIER TO SPOT THE FLOW ON THE PAGE.

	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_5}$	$1_{\heartsuit_0}$	$1_{\clubsuit_0}$	$1_{\diamondsuit_0}$	1_{α_0}
Step 1:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_5}$	$1_{\heartsuit_0}$	$1_{\clubsuit_0}$	$1_{\diamondsuit_0}$	1_{α_0}
	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_4}$	$0_{\heartsuit_1}$	$0_{\clubsuit_1}$	$0_{\diamondsuit_1}$	0_{α_1}	0_{β_1}	·	·
Step 2:	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_4}$	$1_{\heartsuit_1}$	$1_{\clubsuit_1}$	$1_{\diamondsuit_1}$	1_{α_1}	·	·	·
	·	·	·	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_4}$	$1_{\heartsuit_1}$	$1_{\clubsuit_1}$	$1_{\diamondsuit_1}$	1_{α_1}	·	·	·
	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_3}$	$1_{\spadesuit_3}$	$0_{\heartsuit_2}$	$0_{\clubsuit_2}$	$0_{\diamondsuit_2}$	0_{α_2}	·	·	·	·	·
Step 3:	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_3}$	$1_{\spadesuit_3}$	$1_{\heartsuit_2}$	$1_{\clubsuit_2}$	$1_{\diamondsuit_2}$	·	·	·	·	·	·
	·	·	·	·	·	·	·	·	·	$1_{\spadesuit_3}$	$1_{\spadesuit_3}$	$1_{\heartsuit_2}$	$1_{\clubsuit_2}$	$1_{\diamondsuit_2}$	·	·	·	·	·	·
	·	·	·	·	·	·	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$0_{\heartsuit_3}$	$0_{\clubsuit_3}$	$0_{\diamondsuit_3}$	·	·	·	·	·	·	·	·
Step 4:	·	·	·	·	·	·	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\heartsuit_3}$	$1_{\clubsuit_3}$	·	·	·	·	·	·	·	·	·
	·	·	·	·	·	·	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\spadesuit_2}$	$1_{\heartsuit_3}$	$1_{\clubsuit_3}$	·	·	·	·	·	·	·	·	·
	·	·	·	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$0_{\heartsuit_4}$	$0_{\clubsuit_4}$	·	·	·	·	·	·	·	·	·	·	·
Step 5:	·	·	·	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\heartsuit_4}$	·	·	·	·	·	·	·	·	·	·	·	·
	·	·	·	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\spadesuit_1}$	$1_{\heartsuit_4}$	·	·	·	·	·	·	·	·	·	·	·	·
	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
Step 6:	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	·	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	$1_{\spadesuit_0}$	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
	↑ Bit #4																			

20.12 Conclusions

We have provided a simplified proof the Coe, Tang result that the only divisors at risk are those divisors which six consecutive ones in the fifth through tenth positions. We also explain why the absolute error is bounded by `5e-5`, when dividing two numbers in the standard interval $[1, 2)$. Though we do not know exactly how the error occurred, we do point out that the error might well have been more than an elementary careless accident. It is this author's position that the accident was more sophisticated, and perhaps we should think twice before laughing about the error at Intel's expense.

20.13 Acknowledgement

I owe a great deal to Tim Coe for explaining carefully and patiently the reasoning behind his reverse engineering of the Pentium chip. This paper began life as an extended referee's report for the original version of the Coe, Tang paper [21]. Without their paper and Coe's explanation of the Pentium chip, this work would have been impossible. I also thank Velvel Kahan for many interesting discussions during January of 1995, while I was visiting Berkeley, and Vaughan Pratt for reviewing early drafts of this during June of 1995 while I was visiting Stanford. Finally, I thank Teddy Slottow for creating the beautiful color figure.

Chapter 21

When isn't x * (1/x)= 1?

21.1 The Problem

During my very first lecture, I now teach advanced students of numerical analysis the subtleties and intricacies of the IEEE standard for floating point computation [?, ?] These students are then ready to attack the first problem set at the very end of which may be found:

> PROBLEM: Find any IEEE double precision floating point number $1 < x < 2$ such that `x*(1/x)` does not yield 1 exactly. Find the smallest one either by a brute force search or by pruning the search substantially using mathematics. (The typewriter font here denotes a computation as it would be presented to a computer.)

The problem of finding the smallest one is surprisingly difficult. The students have just learned that the double precision numbers between 1 and 2 may be represented as

$$x = 1 + k\epsilon, \quad k = 1, 2, \ldots, 2^{52} - 1, \quad \epsilon = 2^{-52}.$$

Their natural inclination is to perform an incremental search on the computer counting from $k = 1$. This approach can work, but seems to take about a day on the students' workstations to reach the smallest k. As machines get faster, if precision stays the same, the problem may become too trivial, but so far only a few students found the smallest k by brute force. My class consists of graduate students of mathematics, computer science, and engineering. In the 1993 class, nobody succeeded in pruning the search space using mathematics. In the 1994 class, two students Ioanid Rosu and Dimitriy Betaneli succeeded. This note is a simplification of my original solution from 1991 and their solutions.

21.2 The IEEE Standard

We begin by pointing out that the interval $[1, 2]$ fixes a convenient choice for the exponent. Floating point numbers are uniformly spaced in this interval with gap $\epsilon = 2^{-52}$. The interval $[\frac{1}{2}, 1]$ contains as many floating point numbers with gap $\epsilon/2$. We assume that arithmetic is in the "round to nearest mode." For our purposes this specifies that the arithmetic operations of add, subtract, multiply, and divide compute the floating point number nearest to the infinitely precise result. We also need to recall the "round to even" rule which specifies that in case of a tie, i.e., if the infinitely precise result is in the middle of two floating point numbers, the computation produces the one with least significant bit zero.

We use $\mathrm{fl}(z)$ to denote the nearest floating point number to the real number z. Therefore the computation `x*(1/x)` yields the result $\mathrm{fl}(x\mathrm{fl}(\frac{1}{x}))$.

21.3 The Solution

Step 1: The computed value of `x*(1/x)` $\in \{1-\frac{\epsilon}{2}, 1\}$.

Since the gap between consecutive numbers in the interval $(\frac{1}{2},1)$ is $\epsilon/2$, we have that $|\frac{1}{x}-\mathrm{fl}(\frac{1}{x})| \leq \epsilon/4$ which implies that $|1-x\mathrm{fl}(\frac{1}{x})| \leq \epsilon x/4 < \epsilon/2$. Therefore

$$1-\frac{\epsilon}{2} < x\mathrm{fl}(\tfrac{1}{x}) < 1+\frac{\epsilon}{2}$$

from which the result follows by rounding. □

Step 2: The "round to even" rule is never invoked when computing `1/x`.
Proof: The round to even rule would be invoked if and only if $1/x$ were exactly halfway between two floating point numbers, i.e.,

$$\frac{1}{x} = \frac{1}{1+k\epsilon} = 1 - j\frac{\epsilon}{4},$$

where j is odd. This is equivalent to

$$jk = 2^{52}(4k-j).$$

However, it is impossible that jk be a multiple of 2^{52} if j has no even factors and $k < 2^{52}$. □

Step 3: k gives a solution to `x*(1/x)` $\neq 1$, if and only if k is in the interval $(k_-(m), k_+(m))$, where m is an integer,

$$k_-(m) = \frac{1}{4}\left(m+\sqrt{m^2+\frac{8}{\epsilon}(m+\frac{1}{2})}\right), \text{ and } k_+(m) = \frac{1}{4}\left(m+\frac{1}{2}+\sqrt{(m+\frac{1}{2})^2+\frac{8}{\epsilon}(m+\frac{1}{2})}\right).$$

Proof:

Define the integer m by the equation $\mathrm{fl}(\frac{1}{x})= 1-k\epsilon+m\frac{\epsilon}{2}$. Since the gap between numbers in $[\frac{1}{2},1]$ is $\frac{\epsilon}{2}$, we may equivalently define m as the unique integer solution to

$$\left|1-k\epsilon+m\frac{\epsilon}{2}-\frac{1}{1+k\epsilon}\right| < \frac{\epsilon}{4}. \tag{21.1}$$

which implies that

$$(1+k\epsilon)(1-k\epsilon+m\frac{\epsilon}{2}) > 1-\frac{\epsilon}{4}(1+k\epsilon). \tag{21.2}$$

On the other hand, if `x*(1/x)`$\neq 1$ then from Step 1, we must have $x\mathrm{fl}(\frac{1}{x}) < 1-\frac{\epsilon}{4}$ or

$$(1+k\epsilon)(1-k\epsilon+m\frac{\epsilon}{2}) < 1-\frac{\epsilon}{4}. \tag{21.3}$$

Therefore the quadratic inequalities (1),(2) and (3) along with the inequality $k < 2^{52}$ are necessary and sufficient conditions on the positive integers k and m for us to have a situation where `x*(1/x)`$\neq$ 1. Together inequalities (2) and (3) are stronger than inequality (1) so we omit (1). The result is obtained by solving the quadratic inequalities (2) and (3) for k in terms of m. □

Conclusion: A small program reveals that the first m for which $(k_-(m), k_+(m))$ contains an integer is $m = 29$ which gives the answer

$$k = 257,736,490.$$

21.4 Postscript

The recently well-publicized flaw in floating point division on the Pentium chip has prompted the question whether `x/y` may simply be replaced by `x*(1/y)`. This note focuses on the example $y = x$ illustrating that this proposed fix would no longer conform to the IEEE standard.

Chapter 22

Network Topologies

Chapter 23

Topology Based Parallel Algorithms

23.1 The outlawing of graph theory

[This is from my paper – Alan]

I now turn to the issue of communication on the networks of a multicomputer. The issue that I wish to raise here is what constitutes an aesthetically elegant use of network topologies, and by comparison what are the realities of current supercomputing. Perhaps it is best to say that I suffer from an inner conflict between what the mathematician inside of me feels ought to be the way parallel machines should be, and what my engineering colleagues tell me how parallel machines must be, at least for now.

My notion of what a parallel machine for numerical linear algebra ought to be is based on 1) mathematically natural notions, 2) the history of the Basic Linear Algebra Subroutines, and 3) experiences with the CM-2 supercomputer that were never publicized. What each of these three aspects have in common is a reasonably clean abstract world where the user who wishes to use his rational faculties to directly improve the speed of algorithms is allowed to do so. The real world of computing, by contrast, is not so tidy and many experts say that it must not be.

What constitutes natural to a mathematician is easy to explain. Networks consist of interesting topologies which are thought of as graphs, i.e. collections of nodes and edges. The cleanest model of any network allows you to imagine that every time you snap your fingers, a fixed packet of data is allowed to cross exactly one link and all the different links may be used simultaneously. This is the kind of network on which you can do precise mathematics and prove exact theorems. Indeed much very sophisticated work has gone on in this area. A further requirement that I wish to add beyond what is strictly needed for a pure mathematician is that processor address bits or memory address bits (as in the active messages approach devised by Dave Culler of UC Berkeley) need not be sent. These messages are truly couch potato messages in that they are pushed around, rather than actively knowing where they are going or what is going to happen to them whey they get there. This idea is not completely new. One place it has been expressed is Shukla and Agrawal [?].

What ought to seem natural to linear algebra libraries requires understanding how libraries are constructed. LAPACK and its precursors make use of a set of Basic Linear Algebra Subroutines (BLAS) that may be written in a machine specific manner at the assembly language level. It is well understood that the operations in dense linear algebra are so regular and predictable, that it is not unreasonable to sacrifice some people-hours to the task of fine tuning these operations for everyone's benefit. Indeed dense linear algebra is about as regular and predictable as programming can get; it is no wonder that the developers of LAPACK request that manufacturers fine tune the BLAS. The management of registers, cache, and floating point operations is often well enough understood by specialists that optimal or near-optimal management of resources may be obtained.

On a multicomputer, memory is separated so that a communications network must pass messages among the individual processors. Following the lesson from the previous paragraph, it seems natural that there ought to be a collection of Basic Linear Algebra Communications Subroutines (BLACS). It might also seem natural, unless you know too much about current multicomputers, that these BLACS would be written in assembler and would represent a very careful orchestration of communications links. One would expect that the regularity and predictability of dense linear algebra would almost demand such approaches. This was my vision of the BLACS in 1989. Yet the BLACS as they currently exist as part of the new scalable ScaLAPACK have never been written in assembler and few clever mathematical graph theory papers have proven appropriate for these BLACS on any real parallel machines. The BLACS as they exist are calls to a network that will do the work, they do not explicitly manage the network in the sense in which the BLAS manage the registers, cache, and memory.

Engineers say that what I am envisioning can not exist, or if they could exist it would not be useful. There is actual historical evidence that careful control of a machine's network can lead to improvements. Unfortunately, this evidence was never properly understood by network designers. The historical evidence is based on the then somewhat secret successes obtained by scheduling messages on the Connection Machine CM-2 [**?**]. Roughly, the mathematical model of the hypercube described above was not designed into the CM-2, but a number of workers at Thinking Machines beginning with work by Steve Vavasis of Cornell University, showed by using hardware for purposes other than what they were intended for that it was possible to apply the graph theory of the hypercube in the mathematically natural approach described above. Furthermore, this mathematical abstraction was very close to what the machines was really doing. Careful management of the data often lead to 100% utilization of the hypercube channels.

Unfortunately this lesson was lost, partially due to secrecy, partially because Thinking Machines already made the commitment to the CM-5 architecture in which control of the communications network is more limited and less tidy, and finally, maybe, it just is not a good idea. That is for the reader and the future to decide.

The CM-5 contained a random number generator that decides which of two directions a path should take when going up the tree at each level other than the lowest. Thus, any opportunity to control messages is mostly lost other than tricks such as controlling injection into the network. Perhaps this is unfair, but sometimes this seems to me like controlling Boston traffic by putting traffic controls at the city limits.

I admit that I am not a specialist in computer architecture, so I asked some of my colleagues at MIT who are. In the future, I should gather opinions from other institutions as well, but for me this was a small step towards interdisciplinary communication on the issue of communication.

According to Bill Dally at MIT, a good network provides throughput for random traffic that is within 30% of peak capacity. Adaptive routing can increase this to 50% of peak capacity. The belief is that it is far better to devote resources to making the network faster than it is to specialize the control.

Arvind at MIT tends to agree, but he pointed out that exposing the underlying hardware to hard core programmers and compiler writers has often lead to better performance. He is open to the possibility that perhaps a day will come when specialized programmers would have access to the underlying communications protocols though this is not how most programmers would use the machine.

Also at MIT, Steve Ward is explicitly looking at scheduled routing in the NuMesh project.

In conclusion at the current time the BLACS and BLAS are fundamentally different. The BLAS may be written in a careful manner to take advantage of computer memory and orchestrate the data movement between main memory, cache, and registers. The BLACS are currently requests for the network to do the work. Perhaps this is as it must be in the complicated asynchronous world of MIMD supercomputing, even in the regular context of dense linear algebra. Perhaps it must be this way but part of me wishes that it were not so. We will see what the future brings.

23.2 Interprocessor Communication and the Hypercube Graph

As research in parallel computation has come to the forefront of computer science in recent years, parallel algorithm designers have had to struggle with the complications imposed by a large network of processors working to solve a single problem. As the number of processors increases, experimental evidence indicates that *communication bandwidth* (throughput), rather than latency, becomes the bottleneck limiting the scalability of parallel algorithms. Thus, choosing the correct communication sub substrate for one's parallel computer is of paramount importance to its performance. Because of the versatility of an interconnection graph known as the *hypercube*, it looked very attractive to machine designers in the recent past, and was used in the CM-2, Cosmic Cube, and N-Cube (one is located here at MIT, in the Earth Resources Laboratory).

A Hypercube graph consists of:

1. d Dimensions

2. 2^d Vertices

3. $(b_0...b_{d-1})$ and $(b_0^{'}...b_{d-1}^{'})$ are connected by an edge if *and only if they* differ in exactly one bit.

There are $d2^{d-1}$ edges in a hypercube of dimension d.

A picture of the three-dimensional hypercube may not be as helpful as it would seem, but is included as Figure 23.1.

On a sad note, the hypercube design for the most part has been abandoned. The CM-2 allowed control over the network, but no machine since has ever allowed for suchecontrol.

23.2.1 The Gray Code and Other Embeddings of the Hypercube

A hypercube is an amazingly versatile medium on top of which familiar and useful topologies can be easily created. Perhaps the most basic such embedding is the *Hamiltonian Path*[1], a topology in which the nodes appear to be in a ring, so that each node has two neighbors and a message traveling along the ring encounters each node exactly once. Finding such a path in a d-dimensional hypercube is easy because of the existence of a d-bit *Gray code*.

The Gray Code is named after Frank Gray, who formalized it in the 1940's while working on radar-related engineering problems. Given a rapidly changing binary output from a physical measuring device, sampling accuracy can be improved by using the Gray Code to encode adjacent numbers rather than conventional binary. The code was originally invented by Emile Baudet

[1]A Hamiltonian Path is one which touches each node of a graph exactly once. Contrast with an Eulerian Path.

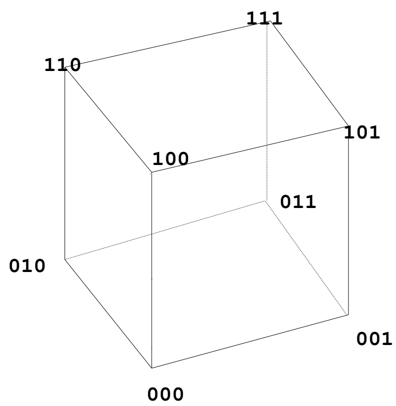


Figure 23.1: The Hypercube for $d = 3$

(1845-1903) for telegraph communication optimization—proper equipment can send several telegraph signals down the same wire to maximize bandwidth under fixed physical constraints.

The Gray Code for $d = 3$ embeds a Hamiltonian path in the three-dimensional hypercube shown above:

GRAY CODE	TRANSITION SEQUENCE
0 0 0	
0 0 1	0
0 1 1	1
0 1 0	0
1 1 0	2
1 1 1	0
1 0 1	1
1 0 0	0

Each time we advance from one element of the Gray Code to the next, we move along a single dimension. The *transition sequence* is merely the sequence of such change directions mapped out as the Gray Code sweeps along its Hamiltonian path. (As a single bit must change at every step, only the dimension of the changed bit, 0, 1, or 2, in this case, must be described.)

To build a Gray Code for d dimensions, one need only take the Gray Code for $d-1$ dimensions, reflect it top to bottom across a horizontal line just below the last element, and add a leading one to each new element below the line of reflection. Given that, you can see how the Gray Code above was derived easily from the Gray Code for $d = 2$:

GRAY CODE (LOCATION OF VERTEX)
0 0
0 1
1 1
1 0

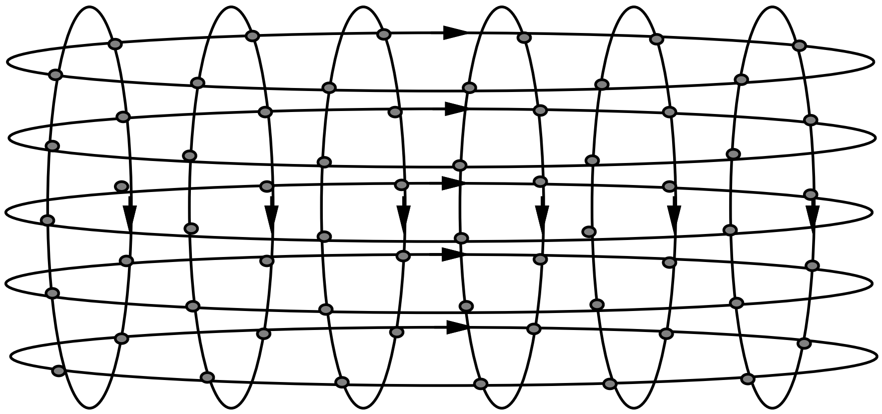


Figure 23.2: Embedding a Two-dimensional Grid in a Hypercube by bit-field partitioning.

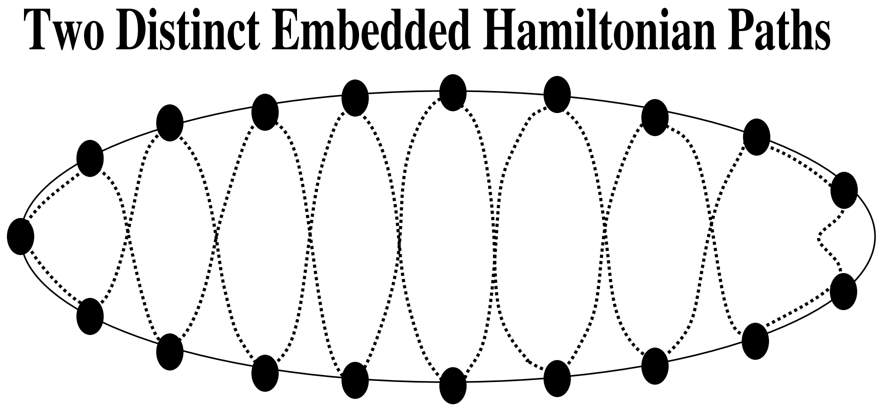


Figure 23.3: Two edge-disjoint Hamiltonian paths spanning a set of nodes.

23.2.2 Going Beyond Embedded Cycles

We are not limited to cycles; now that the Hamiltonian cycle has been described, we can use it to embed a *grid* inside a hypercube. For example, suppose we have a $d = 5$ hypercube. We can group the bits together like $d_4d_3|d_2d_1d_0$, where the left group is given two bits, and the right group three. By using *two independent Gray Codes* for the two sets of bits, we can index into the total set of nodes using *two numbers*. Each number determines a sequential position in one of the Gray Codes (or, equivalently, along one of the Hamiltonian cycles mapped out by them), so that together they completely specify a unique node. By sweeping through the Gray Code mapped to the top two bits, the position moves up and down a "column," and by sweeping through the code mapped to the lower three, the position moves back and forth in a "row". The result is a grid like that shown in Figure 23.2. Three dimensional (or even higher dimensional) grids could be generated using this bit-partitioning technique—we are not limited only to two dimensions.

Finally, note that in a hypercube of high dimensionality, several distinct Hamiltonian paths (spanning the entire set of nodes) can exist which have no (directed) edges in common. A set of nodes can thus be spanned by two completely separate loops around which information can be passed without worrying about collisions. An example of this is shown in Figure 23.3.

23.2.3 Hypercube theory: Unique Edge-Disjoint Hamiltonian Cycles in a Hypercube

Consider in more detail the question of how many edge-disjoint Hamiltonian cycles exist for a hypercube in d dimensions. (If two Hamiltonian cycles traverse an edge in opposite directions, they will be considered edge-disjoint for our purposes.)

For hypercubes of even dimension, there are exactly d edge-disjoint Hamiltonian paths. This is not surprising, since from above we know that a hypercube of dimension d has exactly $d2^{d-1}$ edges. Doubling this number to account for both directions of an edge and dividing it by the number of edges in a Hamiltonian path (2^d, since that's the total number of nodes) *also yields* d. The match tells us that *all edges* can be part of the set of edge-disjoint Hamiltonian paths in a hypercube of even dimensionality.

Strangely, for hypercubes of odd dimension, *there are only* $d-1$ *edge-disjoint Hamiltonian paths.*[2] Since the same edge-use analysis applies as did in the case of hypercubes of even dimension, we conclude some edges must not be used.

We can prove the assertion in the case of $d = 3$. Let's consider a single Hamiltonian cycle embedded in the hypercube of dimension three, as depicted in flattened form in Figure 23.4(a). It should be clear that in three dimensions, any Hamiltonian path can be rotated onto the one shown, so that there is no loss of generality in assuming the presence of the particular path shown. To see this, simply note that each two-dimensional slice of the $d = 3$ cycle must be a cycle in two dimensions (lacking one edge). The cycle in two dimensions is unique except for direction. Furthermore, it can be combined with another such cycle only in one way to form the three-dimensional Hamiltonian cycle. Given this pre-existing cycle in a hypercube of dimension 3, we assume that two additional edge-disjoint Hamiltonian cycles exist in the same hypercube, and produce a contradiction.

The proof is quite simple. Consider in turn the lower-left and upper-left corners of Figure 23.4(a). In the lower-left corner, there are two possibilities for edges entering the corner, and two possibilities for edges leaving. These can be combined to form segments of two paths in a unique way[3], so we have determined the behavior at that point of the two additional edge-disjoint Hamiltonian cycles which we assumed existed. See Figure 23.4(b). A similar analysis can be made for the upper-left corner; we again are led to a unique pair of path segments which must belong to the two new edge-disjoint Hamiltonian cycles, as shown in Figure 23.4(c). The only logical way to connect the segments found in Figures 23.4(b) and 23.4(c) is shown in Figure 23.4(d).

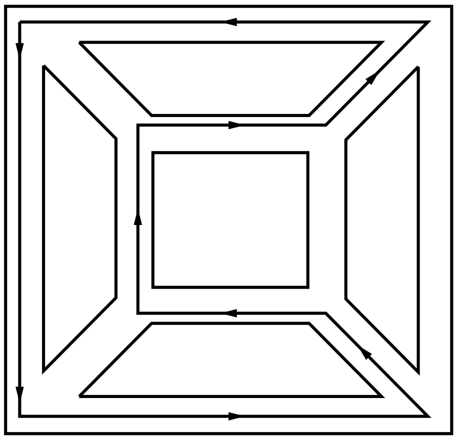
We are almost done. Look at the point marked X on Figure 23.4(d). When the path we have just connected gets to this point, it cannot go to the right, for that edge has already been used for rightward travel by the original Hamiltonian cycle. It certainly can't go back out the way it came in. Thus, it must go down, to the point marked Y. *However, this would form a closed loop consisting of four nodes!* Such a loop might be part of larger loop, but then it wouldn't be *Hamiltonian*, since it would pass through two nodes twice.

Since we have reached a contradiction, we conclude there are not two additional edge-disjoint Hamiltonian paths in the three dimensional hypercube, and in fact only one additional such path. Just reverse the direction of the original Hamiltonian path to find it.

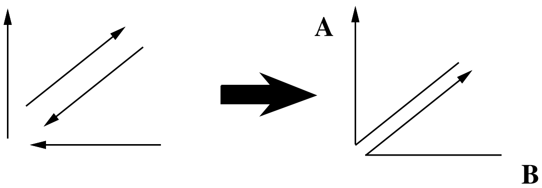
---

[2] I am reminded of the famous vector-field theorem which states that a vector field can exist on the surface of a ball in **n** dimensions without a singular point iff n is even. Could these two results be related?

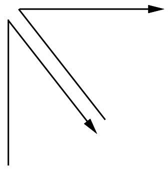
[3] A Hamiltonian cycle in no case immediately backtraces an edge it has just traversed; this provides the uniqueness.



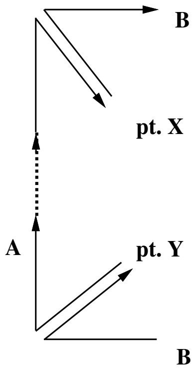
(a) A single Hamiltonian cycle in the d = 3 hypercube



(b) The set of legal incoming and outgoing directed edges at the lower left corner for new Hamiltonian paths



(c) The set of legal Hamiltonian cycle segments at the upper-left corner



(d) Path segment A cannot go right at pt. X because of the original Ham. cycle. But if it goes down it forms a closed loop of 4 nodes!

Figure 23.4: Proof that for the $d = 3$ hypercube there are only 2, not 3, edge-disjoint Hamiltonian cycles.

23.2.4 An Application: Matrix Multiplication on a Grid

The first algorithm for matrix multiplication on a grid is Canon's. This algorithm is designed for two dimensional grids. It still can run very fast on the CM-5 using CSHIFTS, which will be shown later.

The goal is to find **C** where **C** = **A** $*$ **B** and **A**, **B** *and* **C** are all matrices. Canon's algorithm involves two steps: a skewing step and a Systolic step. In the skewing step, the rows of **A** are slid to the left (with wraparound), such that each row is slid to the left a number of slots equal to its row number, with the first row not moving at all. The *columns* of **B** are, analogously, each slid upwards a number of slots equal to their column number, with the first column not moving at all. These movements are an effort to achieve a special configuration of the data which will be used by the following Systolic step. In the systolic step, we perform k iterations, moving **A** horizontally one slot (as a whole) and **B** vertically one slot (also as a whole) in each iteration. As the iterations go by, the products of the elements of **A** and **B** that pass by a location are accumulated into that location, and at the end the product **C** has been computed in each row and column position.

1. Skewing Step
$$A_{i,j} \leftarrow A_{i,i+j} \qquad (modulo\ n)$$
$$B_{i,j} \leftarrow B_{i+j,j} \qquad (modulo\ n)$$
2. Systolic Step
$$C_{(i,j)} =_{k=0}^{n-1} A_{i,j+k} B_{i+k,j}$$

This algorithm is depicted symbolically in Figure 23.5.

Another method for matrix multiplication involves broadcasts on a grid. Thinking about this algorithm on a grid of n^2 processors reduces a matrix multiply to O(n) time (Ref. Bertsekas 77). Imagine a $n * n$ processor machine with a block of matrix **A** and matrix **B** on each node (you can imagine one element on each node, just don't tell Prof. Edelman).

1. Each processor sends $a_{i,j}$ to all the other processors in the i^{th} row. (a_{ij} is the (row i, column j) element of matrix **A**). On a graph this operation looks like Figure 23.6.
2. Each processor sends b_{ij} to the processors in the jth column. This operation is shown in Figure 23.7.
3. The matrix **C** is formed on each node by
$${}_1^n a_{i,j} b_{i,j}$$

This algorithm can also be implemented using C-SHIFTS. This can be easily pictured in Figure 23.6 and Figure 23.7.

Another way of improving Canon's algorithm (systolic step) can be easily pictured. In the original implementation, each element of a block in **A** was sent to the left (to its western neighbor), and each element of a block in **B** was sent up (to its northern neighbor). Thus, in each dimension, the ring of connections was only used unidirectionally. The improvement consists in allowing each

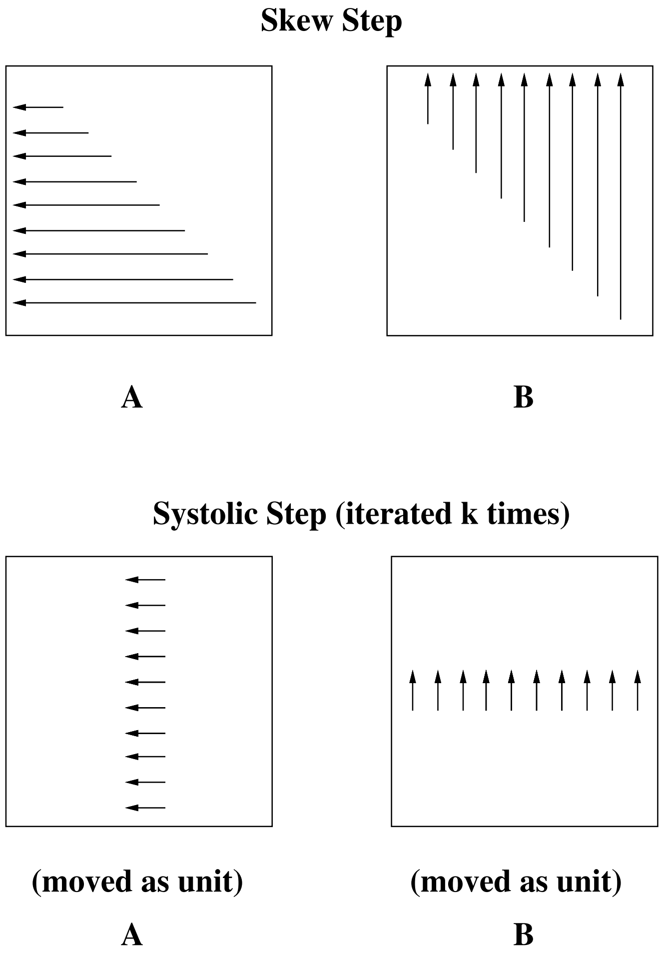


Figure 23.5: Geometry of Canon's Algorithm

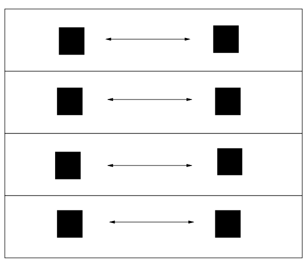


Figure 23.6: Multiple Row Sends

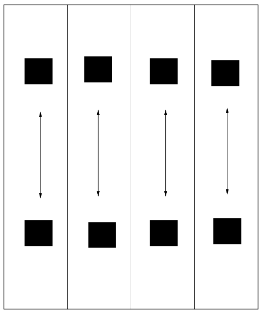


Figure 23.7: Multiple Column Sends

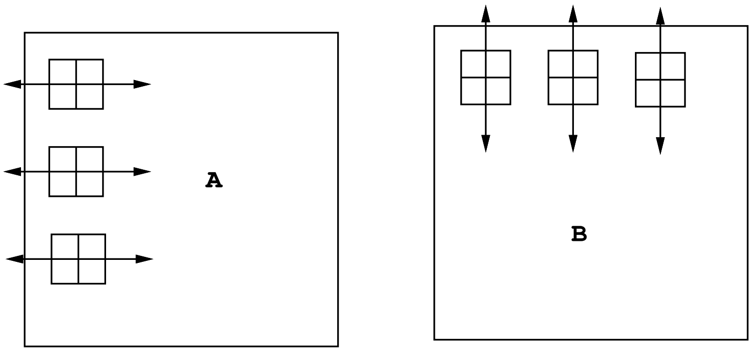


Figure 23.8: Two Direction Matrix Multiply

ring to be used *bidirectionally.* Each block of the **A** matrix can be divided into left and right halves while each block of the **B** matrix can be divided into top and bottom halves. Matrix **A** can simultaneously send information east and west while **B** can simultaneously send information north and south. In such an operation, the elements of a block in **A** in the block's right half are sent east, and the elements of a block in **A** in the block's left half are sent west. Similarly, the elements of a block in **B** in the block's top half are sent north, while the elements of a block in **B** in the block's bottom half are sent south. This does not change the geometry of the information storage in each matrix, and thus the multiplication occurs after much less the communication time. Figure 23.8 describes this operation pictorially.

This method can be best thought of as a ring going in two directions, as is shown in Figure 23.9.

23.2.5 The Direct N-Body problem on a Hypercube

The simplest way to solve the direct N-body problem on a hypercube is two create two copies of the set of bodies, one stationary and one moving, allocated to the nodes spanned by a single large Hamiltonian cycle. In each step, the moving bodies advance along the fixed Hamiltonian path one unit, and interaction computations occur in parallel at each node adding the effects of the current

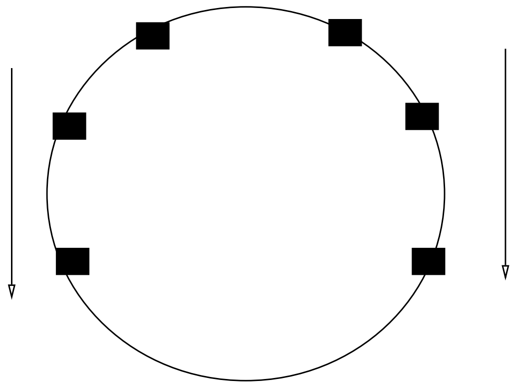


Figure 23.9: Two Direction Ring

pair of bodies to the accumulated sum. To increase efficiency, two edge-disjoint Hamiltonian cycles spanning the nodes of a hypercube can be used instead of a single cycle, as was shown in Figure 23.3. The advantage of this approach is that two sets of moving bodies can be present (each set moving along one of the Hamiltonian cycles) so that the effects of two pairs of bodies are accounted for per systolic step.

A faster method for a direct N-Body solution can take advantage of the hypercube graph and the transition sequence (of a Gray Code) described earlier. A dynamic and static copy of each body can be made at every node. The dynamic body will move around the hypercube changing along the dimension dictated by the transition sequence of a fixed Gray Code. Every body will follow the transition sequence, but not necessarily the Hamiltonian Cycle. The body at (0,0,0) will follow the Hamiltonian cycle exactly, while others will follow in a similar sequence, but not right behind on the same path. Each body will be on a different path, but no body will cross another or "step on another body" at the same moment in time.

In order to define how the bodies other than the one at the zeroth node move, we must introduce the concept of the Hamming sum. (The zeroth node moves exactly in a Hamiltonian path, using the transition sequence derived from that path.)

23.2.6 The Hamming Sum

The Hamming sum will help create an optimal matrix multiply algorithm for the hypercube.

The Hamming Sum is defined as

$$b_{d-1}...b_0 \bigoplus b'_{d-1}...b'_1 b'_0 = b_{d-1} \bigoplus b'_{d-1}, ..., b_1 \bigoplus b'_1, b_0 \bigoplus b'_0$$

where $\bigoplus$ is an *exclusive or* operation. It is essentially just a bitwise exclusive-or operation. (We can now define the **transition sequence** conveniently as the bit position for which $g_i \bigoplus g_{i+1}$ is 1—the dimension along which the Gray Code changes in that step.)

Let's pick a fixed Gray Code G_0. Then the direct N-Body algorithm puts the body that starts at vertex v into the vertex $v \bigoplus g_k$ at the k^{th} step, where g_k is the k^{th} element of G_0.

23.2.7 Extending Gray Code-directed Body Movement to Multiple Sets of Bodies

Heretofore, we have been discussing a *single set of moving bodies.* The one at the zeroth node moves in a Hamiltonian cycle, and the others move in similar paths derived from the original Hamiltonian cycle using the node's position and the Hamming sum. Now, however, we consider *multiple sets of moving objects.* For the sake of argument, let's assume we have a three-dimensional hypercube and three sets of moving objects–a red set, a blue set, and a green set. We want to move each colored set according to the rules just defined for a single set. Obviously, we can't assign G_0 to be the Gray Code for each of the three colors, since that would cause the objects on the zeroth node to travel the exact same Hamiltonian path at the same time and collide. (Assume only a single object can be sent along a wire at one time.) What can be done to use "all of the wires all of the time"?

The solution is to note that there are many Gray Codes for a given bit vector length. New Gray Codes can be created from existing ones by *shifting the columns of bits one or more positions to the left, wrapping around when columns move off the left side.* For example, using our previous table, we could make the transformation:

OLD GRAY CODE	OLD TRAN. SEQ.	NEW GRAY CODE	NEW TRAN. SEQ.
LOC. OF VERTEX		LOC. OF VERTEX	
0 0 0		0 0 0	
0 0 1	0	0 1 0	1
0 1 1	1	1 1 0	2
0 1 0	0	1 0 0	1
1 1 0	2	1 0 1	0
1 1 1	0	1 1 1	1
1 0 1	1	0 1 1	2
1 0 0	0	0 0 1	1

Note that the transition sequences of these two Gray Codes are related in a special way. The second can be produced from the first by adding a 1 (modulo 3) to each element of the first's transition sequence. In fact, there is a third unique Gray Code, which can be produced from the first by adding a *2* (modulo 3) to each element of the first's transition sequence. In general, given a hypercube in $d = k$ dimensions, there are k unique Gray Codes. (Gray codes which represent the same path but traversed in an opposite direction are not counted as unique.)

Thus, if in three dimensions we call these different Gray Codes G_0, G_1, and G_2, we know by how their transition sequences can be derived that, *at any given node of the hypercube*, the transition sequence elements of G_0, G_1, G_2 are all different. Because of this, if we allow one set of dynamic bodies to be controlled by Gray code G_0, another by Gray Code G_1, and the last by Gray Code G_2, we may safely conclude that at any instant in time the red, blue, and green objects that were at the zeroth node at time $t = 0$ follow the Hamiltonian paths implied by G_0, G_1, and G_2 *without colliding.*

You may be asking, *"You proved earlier that you can't have 3 edge-disjoint Hamiltonian paths on a hypercube in three dimensions!? What's going on?"* The question is good one to ask, but is not difficult to answer. Before, we were talking about *edge-disjoint cycles*—paths which never pass down the same edge going the same direction. Here, we're talking about *time-edge-disjoint* cycles. We don't care whether the red, green, and blue objects that were at the zeroth node at $t = 0$ ever

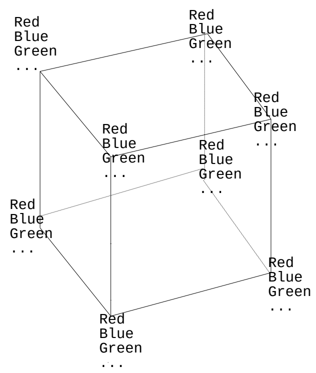


Figure 23.10: N-Body Breakup on a Hypercube

travel down the same wire in the same direction, we only care whether they might do so *at the same time*. As the transition sequences are related by modulo arithmetic in our scheme, a collision is impossible.

Thus, by taking different colored sets of bodies, assigning Gray Codes related by column shifts to those sets, and moving each set around using its assigned Gray Code and the Hamming sum as described above, we can *use all the wires all the time*. In fact, this set up can be used to extend the grid multiplication routine (with the skew and systolic phases) presented earlier so that instead of using only a two-dimensional grid embedded in the hypercube, it uses the hypercube's full connectivity.

23.2.8 Matrix Multiplication using all Hypercube Wires

We can use every connection of the hypercube on every iteration by using the Hamming sum in the calculation of the matrix product. To do so, we require only that the matrices to be multiplied are square, and that the hypercube we are given is of an even dimension. We first split the complete bit-field specifying location in the hypercube into a left and right half, each of which is subsequently addressed as a bit-field of length d/2. Using a Gray Code in d/2 dimensions, g_k, and the hamming sum on d/2-length bit fields, we can perform a matrix multiply in two phases:

1. Skewing Step

$$\mathbf{A}_{ij} \leftarrow \mathbf{A}_{i,i \bigoplus j}$$
$$\mathbf{B}_{ij} \leftarrow \mathbf{B}_{i \bigoplus j,j}$$

2. Systolic Step

$$\mathbf{C} =_k \mathbf{A}_{i,j \bigoplus g_k} \mathbf{B}_{i \bigoplus g_k,j}$$

In each case, when a variable is subscripted by two expressions separated by a comma, the left one refers to the left d/2-bit field, and the right one refers to the right d/2-bit field.

23.2.9 Matrix Multiplication using the Cartesian Product

If $\mathbf{A}$ is any graph with the Cartesian product that allows $\mathbf{k}$ time-wise edge-disjoint cycles, then the Cartesian product of the graph with itself can be used for matrix multiplication. This is essentially a generalization of the idea we used above when we split the original d-bit field of the hypercube given to us into two d/2-bit fields, and formed a Cartesian product of those fields in order to carry out the matrix multiplication in matrix equations immediately above.

23.3 References

[1] Bertsekas, Dimitri, et al. *Parallel and Distributed Computation.* Prentice hall, New Jersey. 1989.

[2] Leighton, F. Thomson. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992.

Parts of these notes were taken from last year's notes written by Scott Wallace.

Chapter 24

Scheduling on Parallel Machines

Chapter 25

Shared Memory Computations

Chapter 26

Scientific Libraries

26.1 Scientific Libraries

A scientific library is a collection of subprograms designed to solve particular mathematical problems. They are written by experts for nonexperts. The expertise involved may be in numerical analysis, in which case the algorithm is the special feature that the writer brings to the software. Good algorithms may run faster and may give better solutions, with less round-off induced inaccuracy. On parallel machines, for the most part, the expertise is in parallelism. The library routine uses a conventional algorithm, but is written in such a way that it works efficiently on a particular parallel architecture. Even better, the library may be written in a portable parallel programming language, such as SPMD code with calls to MPI communication routines; in this case, it runs efficiently on a range of parallel hardware. Special parallel algorithms are sometimes used, but usually the efficiency comes from a clever parallel adaptation of a generally useful algorithm.

For the most part, parallel libraries are in poor shape compared to their sequential siblings. Netlib
(URL `http://www.netlib.org/`) offers a vast collection for sequential machines, very little of it has competitive parallel equivalents. This is because parallel programming is very hard, and until recently there were no good portable parallel programming dialects. There is hope that MPI and HPF will remedy this.

Here is a summary of "what's out there" as of early 1996:

- The ScaLAPACK (or Scalable LAPACK) library
  (URL: `http://www.netlib.org/scalapack/scalapack_home.html`). includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers. It is currently written in a SPMD manner, with message passing (in PVM?). The functionality is at this point a small subset of what the sequential LAPACK library provides.

  An issue in all such work is "where's the matrix?" ScaLAPACK assumes matrices are laid out in a two-dimensional block cyclic manner, since this permits efficient matrix factorizations (see Section **??**.) If the user's matrix is mapped otherwise, that is her problem.

- The Parti/Chaos library
  (URL `http://www.cs.umd.edu/projects/hpsl.html`), which has been developed by Joel Saltz and his colleagues at Yale, ICASE, and Maryland, provides efficient unstructured communication on distributed memory machines, by taking advantage of the fact that many applications make repeated use of the same communication pattern. The communication is

broken into an "inspector" phase, in which bookkeeping operations that depend on a communication pattern but not on the communicated data can be done once and for all, and an "executor" phase in which the communication actually occurs.

- CHACO (`http://www.cs.sandia.gov/HPCCIT/chaco.html`) CHACO is a mesh partitioning library, developed at Sandia National Labs by Bruce Hendrickson and Rob Leland. Chaco has also been applied to a number of problems which have nothing to do with parallel computing, including the determination of genomic sequences, space-efficient circuit placement, organization of databases for efficient retrieval, and ordering of matrices for efficient LU decomposition.

- Petsc

  PETSc, (Portable, Extensible Toolkit for Scientific Computation,
  URL `http://www.mcs.anl.gov/petsc/petsc.html`) is a large suite of data structures and routines for parallel numerical solution of large-scale scientific application problems modeled by partial differential equations, using implicit discretization methods. PETSc 2.0 is usable from Fortran, C, and C++, and runs on most machines. PETSc was developed and is maintained by a group at Argonne National Labs; its principal designers are Bill Gropp and Barry Smith.

- BlockSolve95

  BlockSolve95
  (URL `http://www.mcs.anl.gov/Projects/blocksolve/index.html`) is a scalable parallel software library primarily intended for the solution of sparse linear systems that arise from physical models, especially problems involving multiple degrees of freedom at each node. It is a general purpose iterative solvers for sparse nonsymmetric matrices that have a symmetric nonzero pattern.

  BlockSolve95 runs on a variety of parallel architectures including the the IBM SP series, the Cray T3D, the Intel Paragon, the SGI Power Challenge, and networks of Sun, SGI, DEC alpha, and HP workstations. This portability is obtained by utilizing the MPI message-passing standard. The BlockSolve95 package and the manual can obtained by via anonymous ftp from info.mcs.anl.gov in the directory pub/BlockSolve95, or from netlib.

- Commercially supported libraries for particular parallel machines.

  – IBM's is the PESSL (Parallel Engineering and Scientific Subroutine Library;
    URL `http://lscftp.kgn.ibm.com/pps/vibm/show/products/esslpara.html`).
    PESSL includes BLAS, some linear algebra based on ScaLAPACK, and some FFT routines.

  – Thinking Machines had an impressive library for the CM-5: CMSSL. Unfortunately, much of it seems not to be portable to other architectures; it is available now only to the CM Fortran and CMMD programmer on the CM-5.

  – Maspar had its own library, with some linear algebra, Fourier transform, and image processing kernels. It was written in MPL, a low-level data parallel C dialect for the Maspar. It is not portable.

  – The efforts of some other companies do not extend beyond porting the BLAS.

Bibliography

[1] N. Alon, P. Seymour, and R. Thomas. A separator theorem for non-planar graphs. In *Proceedings of the 22th Annual ACM Symposium on Theory of Computing*, Maryland, May 1990. ACM.

[2] C. R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comp.*, 13(4):932–947, July 1992.

[3] A. W. Appel. An efficient program for many-body simulation. *SIAM J. Sci. Stat. Comput.*, 6(1):85–103, 1985.

[4] I. Babuška and A.K. Aziz. On the angle condition in the finite element method. *SIAM J. Numer. Anal.*, 13(2):214–226, 1976.

[5] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm *Nature*, 324 (1986) pp446-449.

[6] M. Bern, D. Eppstein, and J. R. Gilbert. Provably good mesh generation. *J. Comp. Sys. Sci.* 48 (1994) 384–409.

[7] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In *Computing in Euclidean Geometry*, D.-Z. Du and F.K. Hwang, eds. World Scientific (1992) 23–90.

[8] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. In *Workshop on Algorithms and Data Structures, Springer LNCS 709*, pages 188–199, 1993.

[9] G. Birkhoff and A. George. Elimination by nested dissection. *Complexity of Sequential and Parallel Numerical Algorithms*, J. F. Traub, Academic Press, 1973.

[10] P. E. Bjørstad and O. B. Widlund. Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM J. Numer. Anal.*, 23:1097-1120, 1986.

[11] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT-Press, Cambridge MA, 1990.

[12] J. A. Board, Z. S. Hakura, W. D. Elliott, and W. T. Ranklin. Saclable variants of multipole-based algorithms for molecular dynamic applications. In *Parallel Processing for Scientific Computing*, pages 295–300. SIAM, 1995.

[13] R. Bryant, *Bit-level analysis of an SRT circuit,* preprint, CMU (See `http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/user/bryant/www/home.html`)

[14] V. Carpenter, compiler, `http://vinny.csd.my.edu/pentium.html`.

[15] T. F. Chan and D. C. Resasco. A framework for the analysis and construction of domain decomposition preconditioners. UCLA-CAM-87-09, 1987.

[16] L. P. Chew. *Guaranteed-quality triangular meshes.* TR-89-983, Cornell, 1989.

[17] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems.* North–Holland, 1978.

[18] K. Clarkson, D. Eppstein, G. L. Miller, C. Sturtivant, and S.-H. Teng. Approximating center points with and without linear programming. In *Proceedings of 9th ACM Symposium on Computational Geometry*, pages 91–98, 1993.

[19] T. Coe, Inside the Pentium FDIV bug, *Dr. Dobb's Journal 20* (April, 1995), pp 129–135.

[20] T. Coe, T. Mathisen, C. Moler, and V. Pratt, Computational aspects of the Pentium affair, *IEEE Computational Science and Engineering 2* (Spring 1995), pp 18–31.

[21] T. Coe and P. T. P. Tang, *It takes six ones to reach a flaw,* preprint.

[22] J. Conroy, S. Kratzer, and R. Lucas, Data parallel sparse LU factorization, in *Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 1994.

[23] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1992.

[24] L. Danzer, J. Fonlupt, and V. Klee. Helly's theorem and its relatives. *Proceedings of Symposia in Pure Mathematics, American Mathematical Society*, 7:101–180, 1963.

[25] J. Dongarra, R van de Geijn, and D. Walker, A look at scalable dense linear algebra libraries, in *Scalable High Performance Computer Conference*, Williamsburg, VA, 1992.

[26] I. S. Duff, R. G. Grimes, and J. G. Lewis, Sparse matrix test problems, *ACM TOMS*, 15 (1989), pp. 1-14.

[27] A. L. Dulmage and N. S. Mendelsohn. Coverings of bipartite graphs. *Canadian J. Math.* 10, pp 517-534, 1958.

[28] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3, 193–204, 1986.

[29] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical CS.* Springer-Verlag, 1987.

[30] D. Eppstein, G. L. Miller, and S.-H. Teng. A deterministic linear time algorithm for geometric separators and its applications. In *Proceedings of 9th ACM Symposium on Computational Geometry*, pages 99–108, 1993.

[31] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. Int. J. Num. Meth. Eng. 36:745-764 (1993).

[32] J. Fixx, *Games for the Superintelligent.*

[33] I. Fried. Condition of finite element matrices generated from nonuniform meshes. *AIAA J.* 10, pp 219–221, 1972.

[34] M. Garey and M. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[35] J. A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numerical Analysis*, 10: 345–363, 1973.

[36] J. A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[37] A. George, J. W. H. Liu, and E. Ng, Communication results for parallel sparse Cholesky factorization on a hypercube, *Parallel Comput.* 10 (1989), pp. 287–298.

[38] A. George, M. T. Heath, J. Liu, E. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. on Scientific and Statistical Computing*, 9, 327–340, 1988

[39] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. In *SIAM J. Sci. Comp.*, to appear 1995.

[40] G. H. Golub and C. F. Van Loan. *Matrix Computations, 2nd Edition.* Johns Hopkins University Press, 1989.

[41] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.* 73 (1987) pp325-348.

[42] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[43] R. W. Hackney and J. W. Eastwood. *Computer Simulation Using Particles.* McGraw Hill, 1981.

[44] G. Hardy, J. E. Littlewood and G. Pólya. *Inequalities.* Second edition, Cambridge University Press, 1952.

[45] D. Haussler and E. Welzl. ϵ-net and simplex range queries. *Discrete and Computational Geometry*, 2: 127–151, 1987.

[46] N.J. Higham, The Accuracy of Floating Point Summation *SIAM J. Scient. Comput.* , 14:783–799, 1993.

[47] Y. Hu and S. L. Johnsson. A data parallel implementation of hierarchical N-body methods. Technical Report TR-26-94, Harvard University, 1994.

[48] M. T. Jones and P. E. Plassman. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. *Proc. Scalable High-Performance Computing Conf.* (1994) 478–485.

[49] W. Kahan, *A Test for SRT Division,* preprint.

[50] F. T. Leighton. *Complexity Issues in VLSI.* Foundations of Computing. MIT Press, Cambridge, MA, 1983.

[51] F. T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pp 422-431, 1988.

[52] C. E. Leiserson. *Area Efficient VLSI Computation.* Foundations of Computing. MIT Press, Cambridge, MA, 1983.

[53] C. E. Leiserson and J. G. Lewis. Orderings for parallel sparse symmetric factorization. in *3rd SIAM Conference on Parallel Processing for Scientific Computing*, 1987.

[54] G. Y. Li and T. F. Coleman, A parallel triangular solver for a distributed memory multiproces SOR, *SIAM J. Scient. Stat. Comput.* 9 (1988), pp. 485–502.

[55] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM J. on Numerical Analysis*, 16:346–358, 1979.

[56] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. of Appl. Math.*, 36:177–189, April 1979.

[57] J. W. H. Liu. The solution of mesh equations on a parallel computer. in 2nd Langley Conference on Scientific Computing, 1974.

[58] P.-F. Liu. *The parallel implementation of N-body algorithms.* PhD thesis, Yale University, 1994.

[59] R. Lohner, J. Camberos, and M. Merriam. Parallel unstructured grid generation. *Computer Methods in Applied Mechanics and Engineering* 95 (1992) 343–357.

[60] J. Makino and M. Taiji, T. Ebisuzaki, and D. Sugimoto. Grape-4: a special-purpose computer for gravitational N-body problems. In *Parallel Processing for Scientific Computing*, pages 355–360. SIAM, 1995.

[61] J. Matoušek. Approximations and optimal geometric divide-and-conquer. In *23rd ACM Symp. Theory of Computing*, pages 512–522. ACM, 1991.

[62] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, June 1986.

[63] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. In A. George, J. Gilbert, and J. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, IMA Volumes in Mathematics and its Applications. Springer-Verlag, pp57–84, 1993.

[64] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Finite element meshes and geometric separators. *SIAM J. Scientific Computing*, to appear, 1995.

[65] G. L. Miller, D. Talmor, S.-H. Teng, and N. Walkington. A Delaunay Based Numerical Method for Three Dimensions: generation, formulation, partition. In *the proceedings of the twenty-sixth annual ACM symposium on the theory of computing*, to appear, 1995.

[66] S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. *Proc. 8th ACM Symp. Comput. Geom.* (1992) 212–221.

[67] K. Nabors and J. White. A multipole accelerated 3-D capacitance extraction program. *IEEE Trans. Comp. Des.* 10 (1991) v11.

[68] D. P. O'Leary and G. W. Stewart, Data-flow algorithms for parallel matrix computations, *CACM*, 28 (1985), pp. 840–853.

[69] L.S. Ostrouchov, M.T. Heath, and C.H. Romine, *Modeling speedup in parallel sparse matrix factorization,* Tech Report ORNL/TM-11786, Mathematical Sciences Section, Oak Ridge National Lab., December, 1990.

[70] V. Pan and J. Reif. Efficient parallel solution of linear systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 143–152, Providence, RI, May 1985. ACM.

[71] A. Pothen, H. D. Simon, K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.* 11 (3), pp 430–452, July, 1990.

[72] Vaughan Pratt, *personal communication*, June, 1995.

[73] V. Pratt, Anatomy of the Pentium Bug, *TAPSOFT'95, LNCS 915*, Springer-Verlag, Aarhus, Denmark, (1995), 97–107. `ftp://boole.stanford.edu/pub/FDIV/anapent.ps.gz`.

[74] F. P. Preparata and M. I. Shamos. *Computational Geometry An Introduction.* Texts and Monographs in Computer Science. Springer-Verlag, 1985.

[75] A. A. G. Requicha. Representations of rigid solids: theory, methods, and systems. In *ACM Computing Survey*, 12, 437–464, 1980.

[76] E. Rothberg and A. Gupta, *The performance impact of data reuse in parallel dense Cholesky factorization,* Stanford Comp. Sci. Dept. Report STAN-CS-92-1401.

[77] E. Rothberg and A. Gupta, An efficient block-oriented approach to parallel sparse Cholesky factorization, *Supercomputing* '93, pp. 503-512, November, 1993.

[78] E. Rothberg and R. Schreiber, Improved load distribution in parallel sparse Cholesky factorization, *Supercomputing* '94, November, 1994.

[79] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. *Proc. 4th ACM-SIAM Symp. Discrete Algorithms* (1993) 83–92.

[80] Y. Saad and M.H. Schultz, Data communication in parallel architectures, *Parallel Comput.* 11 (1989), pp. 131–150.

[81] J. K. Salmon. *Parallel Hierarchical N-body Methods.* PhD thesis, California Institute of Technology, 1990. CRPR-90-14.

[82] J. K. Salmon, M. S. Warren, and G. S. Winckelmans. Fast parallel tree codes fro gravitational and fluid dynamical N-body problems. *Int. J. Supercomputer Applications*, 8(2):129–142, 1994.

[83] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, pages 188–260, 1984.

[84] K. E. Schmidt and M. A. Lee. Implementing the fast multipole method in three dimensions. *J. Stat. Phy.*, page 63, 1991.

[85] H.P. Sharangpani and M.L. Barton, *Statistical analysis of floating point flaw in the Pentium ^{TM} Processor* (1994). `http://www.intel.com/product/pentium/white11.ps`

[86] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering* 2:(2/3), pp135-148.

[87] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM J. Scientific Computing*, to appear, 1995.

[88] J. P. Singh, C. Holt, T. Ttsuka, A. Gupta, and J. L. Hennessey. Load balancing and data locality in hierarchical N-body methods. Technical Report CSL-TR-92-505, Stanford, 1992.

[89] G. Strang and G. J. Fix. *An Analysis of the Finite Element Method.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[90] S.-H. Teng. *Points, Spheres, and Separators: a unified geometric approach to graph partitioning.* PhD thesis, Carnegie-Mellon University, School of Computer Science, 1991. CMU-CS-91-184.

[91] V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory Probab. Appl.*, 16: 264-280, 1971.

[92] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. Concurrency, 3 (1991) 457

[93] F. Zhao. An O(n) algorithm for three-dimensional n-body simulation. Technical Report TR AI Memo 995, MIT, AI Lab., October 1987.

[94] F. Zhao and S. L. Johnsson. The parallel multipole method on the Connection Machines. *SIAM J. Stat. Sci. Comp.*, 12:1420–1437, 1991.