Global Partitioning of Parallel Loops and Data Arrays for Caches and Distributed Memory in Multiprocessors

by

Rajeev K. Barua

B.Tech., Computer Science and Engineering
Indian Institute of Technology, New Delhi
(1992)

Submitted to the
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Signature of Author: ______________________________
Department of Electrical Engineering and Computer Science
May 12, 1994

Certified by: ______________________________
A. Agarwal
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: ______________________________
F. R. Morgenthaler
Chairman, Departmental Graduate Committee

Global Partitioning of Parallel Loops and Data Arrays for Caches and Distributed Memory in Multiprocessors

by

Rajeev K. Barua

Submitted to the Department of Electrical Engineering and Computer Science on May 12, 1994 in partial fulfillment of the requirements for the Degree of

Master of Science

in Electrical Engineering and Computer Science

ABSTRACT

This thesis presents a solution to the problem of automatically partitioning loops and arrays for cache-coherent distributed memory multiprocessors. The compiler algorithm described is intended for such machines, though it handles machines without caches as well.

A loop partition specifies the distribution of loop iterations across the processors. A data partition specifies the distribution of arrays. Loops are partitioned in order to get good cache reuse, while data partitioning endeavors to make most array references access the local memory of the processor issuing them. The problems of finding loop and data partitions are related, and must be done together. Our algorithm handles programs with multiple nested parallel loops accessing many arrays with array access indices being general affine functions of loop variables.

We present a cost model which estimates the cost of a loop and data partition given machine parameters such as cache, local and remote access timings. Minimizing the cost as estimated by our model is an NP-complete problem, as is the fully general problem of partitioning. We present a heuristic method which provides solutions in polynomial time.

The scheme has been fully implemented in our compiler for the Alewife machine. We demonstrate the method on several small program fragments, and show performance results on one large application, namely the *conduct* routine in SIMPLE, which has 20 parallel loops (including both one and two dimensional loops) and 20 data arrays, which are shared by several loops.

Thesis Advisor: A. Agarwal

Title: Associate Professor of Computer Science and Engineering

Acknowledgments

The Alewife project at MIT is a collaborative project involving the contributions of many people. It was using the infrastructure developed by these people that this research was possible. Foremost, I like to thank my advisor Anant Agarwal for his guidance and encouragement throughout the research project. He was the one who got me interested in this area in the first place, and was a co-researcher for large parts of the project. He, in effect taught me, a completely inexperienced researcher, how to do research. His advice, in many things major and minor, has been invaluable.

David Kranz has been an active co-researcher in this work. Indeed, most of the work in this thesis has been the result of our joint labor. His involvement throughout the design and implementation stages of the project have been invaluable. It is safe to say that without his help and deep knowledge of the existing system, this work would not have been possible. The many brainstorming sessions that I have had with Anant and David (sometimes both together) helped initiate many of the ideas in this work. In some ways, the whole experience of learning to do research was perhaps more valuable to me than the work itself.

Another person who I would like to thank is Venkat Natarajan of Motorola (Cambridge). He has also been interested in this area, and the work in this thesis builds up on the work he did with Anant and David before I joined the group. In addition I spent last summer at Motorola working with him on an area very closely related to this. His ideas and enthusiasm have been a great help.

Many other people have been a help during this project. Kirk Johnson, David Chaiken, Beng-Hong Lim and John Kubiatowicz have always been ready to answer my queries in various areas, and have helped fix bugs and write new routines for the software for the compiler, simulator and kernel, whenever we reported bugs or asked for additional functionality. Beng helped us provide a scan function in the compiler. I thank my helpful officemates, David and Beng for their help and advice on many occasions.

A special thanks to my apartment mates Nagi, Arvind and Naresh for providing a great family-like environment at home. They and my other friends helped me get through the times I was most stressed out, especially a period in October, and then again in February/March, when the research work was the most hectic. Finally, I thank my parents and sisters for the support and encouragement they give in many ways.

Contents

List of Figures

Chapter 1

Introduction

The problem of loop and data partitioning for distributed memory multiprocessors with global address spaces has been studied by many researchers [1, 2, 4, 14]. The goal of *loop partitioning* for applications with nested loops that access data arrays is to divide the iteration space among the processors to get maximum reuse of data in the cache, subject to the constraint of having good load balance. For architectures where non-local memory references are more expensive than local memory references, the goal of *data partitioning* is to place data in the memory where it is most likely to be accessed by the local processor. Data partitioning tiles the data space and places the individual data tiles in the memory modules of the processing nodes.

In this work the interaction between loop and data partitioning is focused on. If a loop is partitioned in order to get good data reuse in the cache, that partition determines which processor will access each datum. In order to get good data locality, the data should be distributed to processors based on that loop partition. Likewise, given a partitioning of data, a loop should be distributed based on the placement of data used in the loop. This introduces a conflict when there are multiple loops because a loop partition may have two competing constraints: good cache reuse may rely on one loop partition being chosen, while good data locality may rely on another.

1.1 A Cost Model

In order to obtain a global partitioning of all loops and data in a program I introduce a cost model that estimates the cost of executing a loop given the loop partitions and the partitions of data arrays accessed by the loop. This cost model is based on architectural parameters such as the cost of local and remote cache misses. The cost model is used to drive an iterative solution to the global problem.

1.2 Overview of the Algorithm

This thesis presents a solution to the problem of determining loop and data partitions automatically for programs with multiple loops and data arrays. We assume that parallelism in the source program is specified using parallel **do** loops. This can either be done by a programmer, or by a previous dependence analysis and parallelization phase. The algorithm presented is mainly directed towards cache-coherent multiprocessors with physically distributed memory.

Initially, the basic algorithm for deriving loop partitions defined in [2] is used. These partitions are optimized for cache reuse without regard to data locality. This partition is used as an initial loop partition and the induced data partition is used as the initial data partition. That is, we partition each array in the same way as the largest loop that accesses that array. It may then be the case that there are loops accessing multiple arrays that have a large number of remote references. It might be better to re-partition such loops to increase data locality. This re-partition would be at the expense of cache reuse; the better partition is determined by architectural parameters using the cost model, which thus controls the heuristic search of the space of global loop and data partitions.

The above outlines one iteration of the algorithm. Successive iterations are run to improve upon the solution produced, with each iteration composing two phases: a forward phase followed by a back phase. The forward phase finds the best data partitions given the current loop partitions, and the back phase possibly changes the loop partitions given the newly determined data partitions.

1.3 Method of Investigation

The heuristic search method presented in this work has been implemented in our compiler for the MIT Alewife machine and some results to find its effectiveness have been collected. We will describe the implementation and present some results using a part of the SIMPLE program. The NWO simulator [7] for the Alewife machine has been used in this process. We present the improvement in locality achieved as well as the impact on overall performance.

1.4 Overview of Results

It was found that the heuristic method provided significant speedup over using the method described in [2], which itself had significant speedup over using random data partitions with cache-optimized loop partitions. Further, it was found that for a machine like Alewife in which remote accesses are much more expensive than local accesses, the benefit of cache locality optimization is small, relative to that of data locality optimization. However, by changing the architectural parameters input to the cost model, the benefit of cache optimization was seen to increase as local access time became closer to remote access time. In the extreme case of the two being equal (uniform memory access machines), the cost model predicted a benefit for cache optimization only, as expected, while data partitioning became irrelevant.

1.5 Overview of the Rest of the Thesis

The rest of the thesis is organized as follows. Chapter 2 describes related work. Chapter 3 describes the framework and notation used by [2], which we build on. Chapter 4 describes the cost model. Chapter 5 describes the heuristic method. Chapter 6 looks at the relative benefits of, and the tradeoffs between, optimizing for cache and data locality. Chapter 7 describes some experimental results. Chapter 8 concludes and summarizes the thesis, and looks at possible future work.

Chapter 2

Related Work

The problem of data and loop partition has been looked at by many researchers. One approach to solve this problem is to leave it to the user to specify data partitions explicitly in the program, as in Fortran-D [10, 16]. Loop partitions are usually determined by the owner computes rule. Though simple to implement, this requires the user to thoroughly understand the access patterns of the program, a task which is not trivial even for small programs. For real medium-sized or large programs, the task is a very difficult one. Presence of fully general affine function accesses further complicates the process. Further, the user would need to be familiar with machine architecture and architectural parameters to understand the trade-offs involved.

Ramanujam and Sadayappan [14] deal with data partitioning in multicomputers and use a matrix formulation; their results do not apply to multiprocessors with caches. Their theory produces communication-free hyperplane partitions for loops with affine index expressions when such partitions exist. However, when communication-free partitions do not exist, they deal only with index expressions of the form variable plus a constant.

Abraham and Hudak [1] look at the problem of automatic loop partitioning for cache locality only for the case when array accesses have simple index expressions. Their method uses a local per-loop analysis.

A more general framework was presented by Agarwal et. al. [2] for optimizing for cache locality. They handled fully general affine access functions, i.e. accesses of the form A[2i+j,j] and A[100-i,j] were handled. We borrow the concept of uniformly generated references from their work, which was used earlier in Wolf and Lam [17] and Gannon et. al. [8] also. However, they found local minima for each loop independently, giving possibly conflicting data partitioning requests across loops.

The work of Anderson and Lam [4] does a global analysis across loops, but has the following differences with our method: (1) It does not take into account the effect of globally coherent caches. Many new multiprocessors have this feature. (2) It attempts to find a communication free partition by satisfying a system of constraints, failing which it resorts to rectangular blocking. Our method of having a cost model can evaluate different competing alternatives, each having some amount of communication, and choose between them. (3) We guarantee a load balanced solution.

Gupta and Banerjee [9] have developed an algorithm for partitioning doing a global analysis across loops. They allow simple index expression accesses of the form $c_1 * i + c_2$, but not general affine functions. They do not allow for the possibility of hyperparallelepiped data tiles, and do not account for caches.

The work of Wolf and Lam [18] complements ours. They deal with the problem of taking sequential loops and applying transformations to them to convert them to a set of parallel loops with at most one outer sequential loop. This technique can be used before partitioning when the programming model is sequential to convert to parallel loops.

Chapter 3

Loop Partitioning Overview

This chapter gives a brief summary of the method for loop partitioning to increase cache reuse given in [2]. We use this method as the starting point for loop and data partitioning. The method handles programs with loop nests where the array index expressions are affine functions of the loop variables. In other words, the index function can be expressed as,

$$\vec{g}(\vec{i}) = \vec{i}\mathbf{G} + \vec{a} \tag{3.1}$$

where $\mathbf{G}$ is a $l \times d$ matrix with integer entries and $\vec{a}$ is an integer constant vector of length d, termed the *offset vector*. $\vec{i}$ is the vector of loop variables, and $\vec{g}(\vec{i})$ is the vector of array index expressions. Thus affine functions are those which can be expressed as linear combinations of loop variables plus a constant. For example, accesses of the form A[2i+j,100-i] and A[j] are handled, but not A[i^2], where i,j are nested loop induction variables.

3.1 Footprints and Uniformly Intersecting Sets

A loop partition $\mathbf{L}$ is defined by a hyperparallelepiped at the origin as pictured in Figure 3.1. The number of iterations contained in $\mathbf{L}$ is $|\det \mathbf{L}|$. The *footprint* of a loop tile $\mathbf{L}$ with respect to an array reference is the set of points in the data space accessed

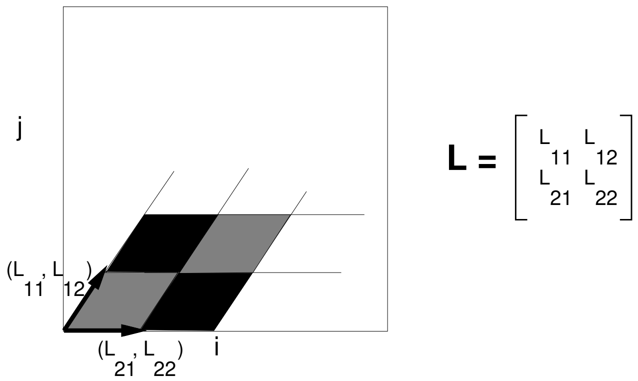


Figure 3.1: Iteration space partitioning is completely specified by the tile at the origin.

by the loop tile as a result of the reference. This footprint is given by **LG**. A set of references with the same **G** but different offsets are called *uniformly intersecting* references. The footprints associated with such sets of references are the same shape, but are translated in the data space. This can be illustrated using the following code fragment.

Doall `(i=0:99, j=0:99)`

`A[i,j] = B[i+j,j]+B[i+j+1,j+2]`

EndDoall

This code has two uniformly intersecting references for array B and a **G** matrix given by

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Assume that the loop tile at the origin **L** is given by

$$\begin{bmatrix} L_1 & L_1 \\ L_2 & 0 \end{bmatrix}.$$

Figure 3.2 shows this tile at the origin of the iteration space and the footprint of the tile (at the origin) with respect to the reference $B[i+j, j]$ is shown in Figure 3.3. Finally, Figure 3.4 shows the combined footprints of both references.

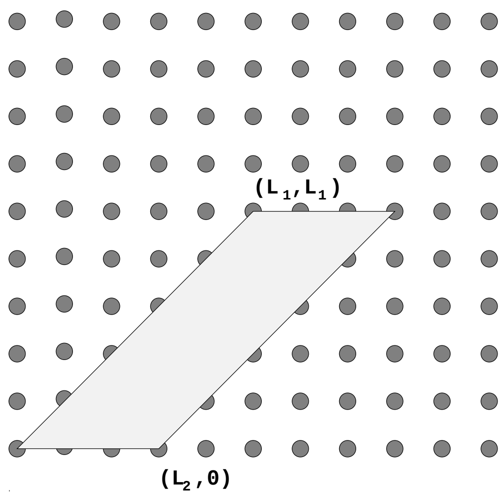


Figure 3.2: Tile **L** at the origin of the iteration space.

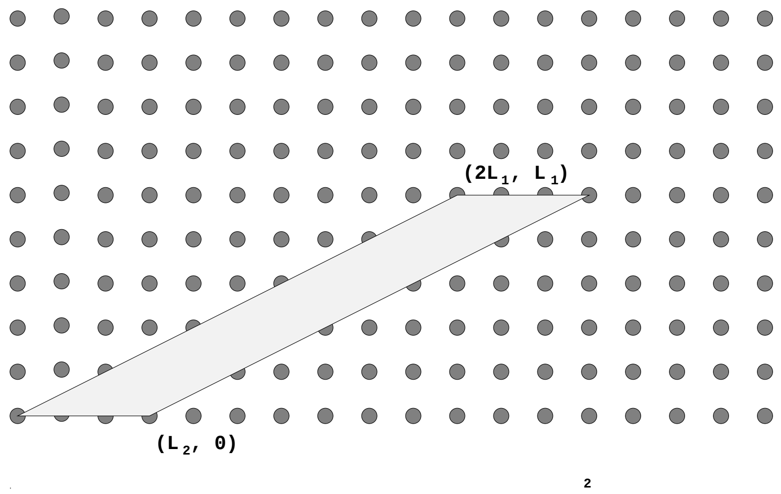


Figure 3.3: Footprint of **L** wrt $B[i+j, j]$ in the data space.

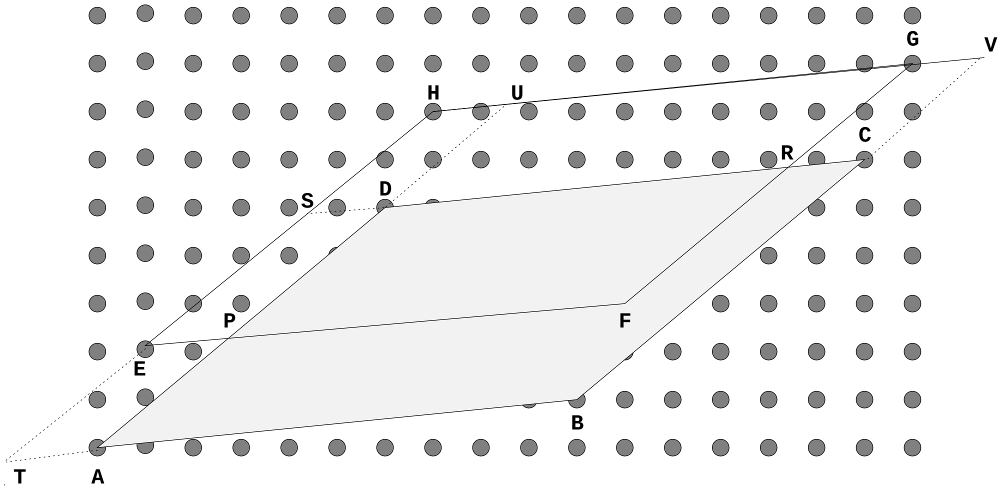


Figure 3.4: Data footprint wrt $B[i+j,j]$ and $B[i+j+1,j+2]$

3.2 Cumulative footprint minimization

The objective of cache locality optimization is to have maximum reuse of data in the cache. If the total number of references in the program loops is fixed, then this is achieved by minimizing the sum of the cumulative footprints of all the arrays. This is because all data elements in the cumulative footprints have only their first time access not in cache, assuming a large enough cache, and no interference. [2] shows how $\mathbf{L}$ can be chosen to minimize the number of cache misses. In doing so, it shows how the combined footprints of a set of uniformly intersecting references can be characterized by a single offset vector $\hat{a}$. This vector is used in the cost model presented in the next chapter. The vector in a sense is the summary of all the offset vectors in a uniformly intersecting set. [2] presents a theorem giving the size of the cumulative footprint, which we reproduce here:

Theorem 1 *Given a hyperparallelepiped tile* $\mathbf{L}$ *and a unimodular reference matrix* $\mathbf{G}$*, the size of the cumulative footprint with respect to a set of uniformly intersecting references specified by the reference matrix* $\mathbf{G}$ *and a set of offset vectors* $\vec{a_1}, ..., \vec{a_R}$*, is approximately*

$$| \; Det \; \mathbf{LG}| + \sum_{k=1}^{d} | \; Det \; \mathbf{LG}_{k \to \hat{a}}|$$

where $\hat{a} = spread_{\mathbf{LG}}(\vec{a_1}, ..., \vec{a_R})$ *and* $\mathbf{LG}_{k\rightarrow\hat{a}}$ *is the matrix obtained by replacing the kth row of* $\mathbf{LG}$ *by* $\hat{a}$.

The above yields an expression for the cumulative footprint for one loop for each array. The sum of these expressions for all the arrays in a loop yields the total cumulative footprint of the loop. This can then be minimized by conventional methods to obtain the loop partitioning with best cache locality. A complete description of this technique appears in [2].

In the cost model we also refer to the data partition $\mathbf{D}$. $\mathbf{D}$ represents how the data space is tiled. This is represented as a tile at the origin of the data space just like $\mathbf{L}$ is represented as a tile at the origin of the iteration space. An array reference in a loop will have good locality when $\mathbf{LG} = \mathbf{D}$.

3.3 An example

This section presents an example of cache locality optimization using the above method. Consider the following code fragment.

```
Doall (i=0:N, j=0:N)
  A[i,j] = B[i+1,j]+B[i,j+2]
EndDoall
```

There are two uniformly intersecting classes of references, one for array A, and one for B. Because A has only one reference, its footprint size is independent of the loop partition, given a fixed total size of the loop tile, and therefore need not figure in the optimization process.

We may in our implementation choose to restrict loop tiles to be rectangles and allow data tiles to be hyperparallelepipeds in general. This is not a serious loss in flexibility as either loop or data tiles being hyperparallelepipeds allows programs with affine function accesses to match their loop and data tiles well. In our implementation, since its easier and more efficient to implement hyperparallelepiped data tiles than loop tiles (for reasons not mentioned here), we make this choice.

Thus, assuming that the loop tile **L** is rectangular, it is given by

$$\begin{bmatrix} L_1 & 0 \\ 0 & L_2 \end{bmatrix}.$$

Because **G** for the references to array B is the identity matrix, the $\mathbf{D} = \mathbf{LG}$ matrix corresponding to references to B is the same as **L** , and the $\hat{a}$ vector is *spread*((1,0),(0,2)) = (1,2). Thus the size of the corresponding cumulative footprint according to theorem 1 is

$$\begin{vmatrix} L_1 & 0 \\ 0 & L_2 \end{vmatrix} + \begin{vmatrix} 1 & 2 \\ 0 & L_2 \end{vmatrix} + \begin{vmatrix} L_1 & 0 \\ 1 & 2 \end{vmatrix}.$$

The size of this cumulative footprint reduces to $L_1L_2 + 2L_1 + L_2$. We minimize this subject to the constraint that the area of the loop tile, $|\text{ Det } \mathbf{L}|$, is a constant to ensure a balanced load. For example if the loop bounds are I, J then the constraint is $|\text{ Det } \mathbf{L}| = IJ/P$, where P is the number of processors.

The optimal values for L_1 and L_2 can be shown to satisfy the equation $2L_1 = L_2$ using the method of Lagrange multipliers. Using actual values of P and the loop bounds, they are solved for exactly.

Chapter 4

The Cost Model

In order to evaluate the combined cache and data locality of loop and data partitions we use a cost function to compare different solutions. This function takes, as arguments, a loop partition, data partitions for each array accessed in the loop, and architectural parameters that determine the relative cost of cache misses and remote memory accesses. It returns an estimation of the cost of array references for the loop.

4.1 Derivation of the formula

In this section we shall define the cost formula in terms of the variables used to define partitions in [2]. We shall begin in section 4.1.1 by stating a basic formula for total access time in a loop, in terms of the number of cache, local and remote accesses in the loop. Then in section 4.1.2, we shall define certain functions, with a view to using them to express the total number of accesses to different levels of memory in terms of the variables used to define partitions. Finally in section 4.1.3 we shall show how the number of accesses to different levels of memory can actually be expressed in terms of these functions.

4.1.1 A basic formula

We can express the cost due to memory references in terms of architectural parameters in the following equation:

$$T_{total\_access} = T_R(n_{remote}) + T_L(n_{local}) + T_C(n_{cache})$$

where T_R, T_L, T_C are the remote, local and cache memory access times respectively, and $n_{remote}, n_{local}, n_{cache}$ are the number of references that result in hits to remote memory, local memory and cache memory. T_C and T_L are fixed by the architecture, while T_R is determined both by the base remote latency of the architecture and possible contention if there are many remote references. T_R may also vary with the number of processors based on the interconnect topology.

n_{cache}, n_{local} and n_{remote} depend on the loop and data partitions. The task we have is to define these three variables in terms of the notation presented for partitions defined in Chapter 3. To do this exactly is a very hard problem. However, with an approximation which is almost always true, it is tractable. We show how in the remainder of this chapter.

4.1.2 Some Definitions

We define the functions R_b, F_f and F_b, which are all functions of the loop partition L, data partition D and reference matrix G with the meanings given in Chapter 3. For simplicity, we also use R_b, F_f and F_b, to denote the value returned by the respective functions for the partition being evaluated.

In order to define these functions, we need the concept of a base offset for each uniformly intersecting set. A uniformly intersecting set may contain several references, each with slightly different footprints due to different offsets in the array index expressions. We pick one and call it the base offset, or $\vec{b}$. Whenever we induce a data partition from a loop partition, we shall make the data partition coincide with the footprint of the base offset reference in the uniformly intersecting set.

Definition 1 *R_b is a function which maps L, D and G to the number of remote references that result from a single index expression defined by G and the base offset $\vec{b}$, in accessing D.*

In other words, R_b returns the number of remote accesses that result from a single program reference in a parallel loop, where the data partitioning is specified by D.

To simplify the computation of R_b we make an approximation: We assume that one of the two following cases apply to loop and data partitions.

1. That loop partition L matches the data partition D perfectly, in which case we assume that $R_b = 0$,

2. or that L does not match D, in which case we assume all references are remote.

The latter case results in the maximum value of R_b. Note that L matches D if LG = D and the origin of D is a mapping of the origin of L with the given access function. That is:

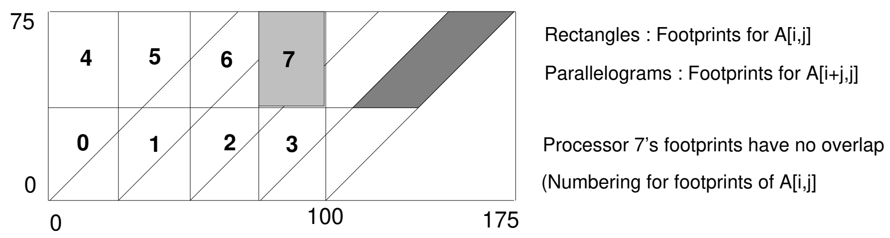
$$\begin{aligned} R_b &= 0 && \text{if LG = D and } Origin(D) = Origin(L)G + \vec{b} \\ &= |\text{Det L}| && \text{otherwise} \end{aligned}$$

where, $Origin(D)$ is the offset vector of the data tile in the data space at processor zero, and $Origin(L)$ is the offset vector of the loop tile in the iteration space at processor zero. $Origin(L)G + \vec{b}$ is the mapping of the loop tile offset vector into the data space via the access matrix G.

The reason this is a good approximation is that both L and D represent regular tiling of their spaces. This means that if L and D partially overlap at the origin, there will be less overlap on other processors. For a reasonably large number of processors, some will end up with no overlap as shown in the example in Figure 4.1. Since the

```
Doall (i=0:100, j=0:75)
   B[i,j] = A[i,j] + A[i+j,j]
EndDoall
```

(a) Code fragment



(b) Data space for Array A(8 processors)

Figure 4.1: Different uniformly intersecting sets have no overlap

execution time for a parallel loop nest is limited by the slowest processor, it will be the same as if all processors had no overlap. Hence, this is a very good approximation.

Definition 2 *F_b is the number of first time accesses in the footprint of L with base offset $\vec{b}$. Hence:*

$$F_b = |Det\ L|$$

Definition 3 *F_f is difference between (1) the cumulative footprints of all the references in a given uniformly intersecting set for a loop tile, and (2) the base footprint due to a single reference represented by G and the base offset $\vec{b}$. We refer to F_f as the peripheral footprint.*

Although the exact details of how F_f is computed is not important to understanding the heuristic method in the next chapter, [2] demonstrates that F_f can be computed as:

$F_f = \sum_{k=1}^{d} \mid Det\ D_{k \rightarrow \hat{a}} \mid$

where $\hat{a}$ is the spread vector of references as mentioned in Chapter 3, and defined in [2].

4.1.3 The Final Formula

Theorem 2 *The cumulative access time for all accesses in a loop with partitioning L, accessing an array having data partition D with reference matrix G in a uniformly intersecting set, is*

$$T_{total\_access} = T_R(R_b + F_f) + T_L(F_b - R_b) + T_C(nref - (F_f + F_b))$$

where $nref$ is the total number of references made by a loop tile for the uniformly intersecting set.

This result can be derived as follows. The number of remote accesses n_{remote} is the number of remote accesses with the base offset, which is R_b, plus the size of the peripheral footprint F_f, giving $n_{remote} = R_b + F_f$. The number of local references n_{local} is the base footprint, less the remote portion, i.e. $F_b - R_b$. The number of cache hits n_{cache} is clearly $nref - n_{remote} - n_{local}$ which is equal to $nref - (F_f + F_b)$.

4.2 Limitations

The cost model is incomplete in the following two ways. A complication arises when one uniformly intersecting set influences another. This may happen when two loops access the same array with the same reference matrix G resulting in n_{cache} to be higher than predicted above. This is because much of the data may be in the cache due to the earlier loop when we execute the later loop.

Also, a linear flow of control through the loop nests of the program has been assumed. While this is a common case, conditional control flow still should be handled.

Although this case is not handled now, we intend to handle this by assigning probabilities to each loop nest, perhaps based on profile data. This probability can then be multiplied by the loop size to get an effective loop size for the algorithm to use.

I have not yet implemented solutions to these two problems as yet.

Chapter 5

The Multiple Loops Heuristic Method

In this chapter we present the heuristic search algorithm for finding loop and data partitions. We begin by stating the bipartite graph data structure we use in the optimization process in section 5.1. Next we present the method itself in section 5.2. Section 5.2.1 demonstrates the method by an example, and section 5.2.2 presents certain details of the method. Finally section 5.3 shows that the algorithm has polynomial time complexity.

5.1 Graph formulation

A commonly used natural representation of a program with many loops accessing many arrays is a bipartite graph G $= (V_l, V_d, E)$, as in [4]. We picture the loops as a set of nodes V_l on the left hand side, and the data arrays as a set of nodes V_d on the right. An edge e$\in$E between a loop and array node is present if and only if the loop accesses the array. The edges are labeled by the uniformly intersecting set(s) they represent.

5.2 Iterative Method

The basic method is the following. We begin with an initial loop partition arrived at by the single loop optimization method described in Chapter 3. Then we follow an iterative improvement method with each iteration having two phases: the first (forward) phase finds the best data partitions given loop partitions, and the second (back) phase determines the values of the loop partitions again, given the data partitions just determined.

Specifically, in the forward phase we set the data partition of each array to be one of the data partitions induced by the loops accessing it. The choice is to pick the data partition induced by the largest loop accessing it. There are some details and changes to this basic strategy which we discuss in Section 5.2.2. In the back phase we set the loop partition of each loop to be one of the loop partitions induced by the arrays accessed by the loop. We use the cost model to evaluate the alternative loop partitions and pick the one with the minimum cost.

These forward and backward phases are repeated, each time using the cost model to determine the estimated array reference cost for the current partitions. After some number of iterations, the best partition found so far is picked as the final partition. Termination is discussed in Section 5.2.2. In practice this heuristic seems to perform quite well.

5.2.1 An example

We demonstrate the workings of the heuristic on a simple example. Consider the following code fragment:

```
Doall (i=0:99, j=0:99)
  A[i,j] = i * j
EndDoall
Doall (i=0:99, j=0:99)
```

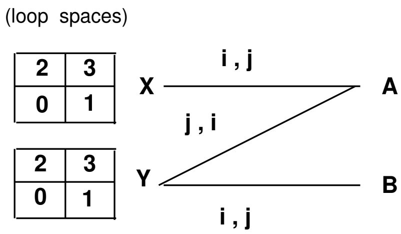


Figure 5.1: Initial solution to loop partitioning (4 processors)

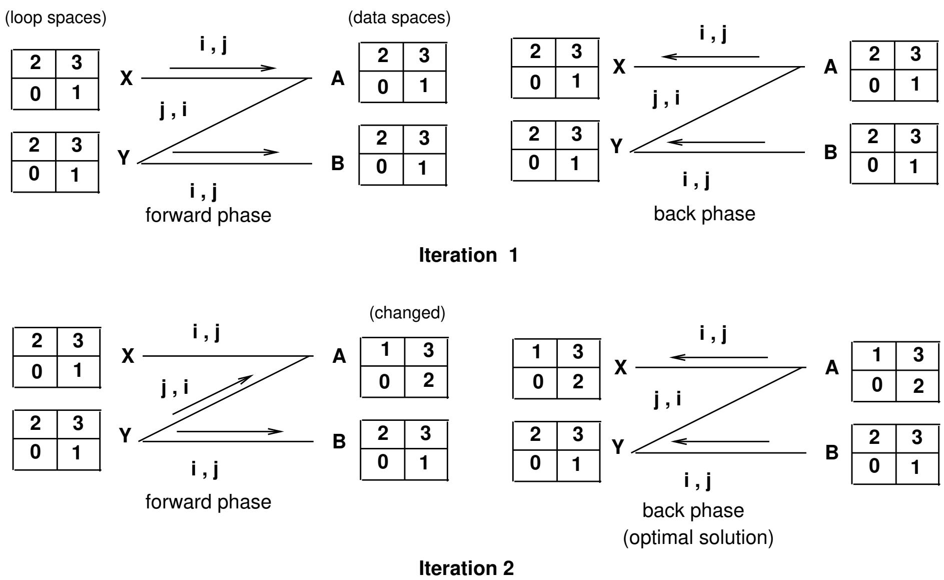


Figure 5.2: Heuristic: iterations 1 and 2 (4 processors)

```
B[i,j] = A[j,i]
```

EndDoall

The code does a transpose of A into B. The first loop is represented by X and the second by Y. The initial cache optimized solution for 4 processors is shown in Figure 5.1. In this example, as there is no peripheral footprint for either array, a default load balanced solution is picked. Iterations 1 and 2 with their forward and back phases are shown in Figure 5.2.

In iteration 1's forward phase A and B get data partitions induced by their largest accessing loops. Since both loops here are equal in size, the compiler picks either, and one possible choice is shown by the arrows. In 1's back phase, loop Y cannot match both A and B's data partitions, and the cost estimator indicates that matching either has the same cost. So an arbitrary choice as shown by the back arrows induces unchanged data partitions. So until now, nothing has changed from the beginning.

In iteration 2, something more interesting happens. As explained in the next section, the choice of data partitions to be induced in the forward phase is *favored in the direction of change*. So now array A picks a different data partition from before, induced by Y instead of X. In the back phase loop X now changes its loop partition to reduce cost as dictated by the cost estimator. This is the best solution we find, and no further change is induced in subsequent iterations. In this case, this best solution is also the optimal solution as it has 100% locality.

The choices we made for tie breaking when costs were equal and for inducing forward iterations were worst case: any other choice would have a solution in the same or lesser time. For example, if in iteration 1's forward phase array A's data partition was induced by Y instead of X, we would find the same solution in iteration 1's back phase itself. Finally, we note that in this example a communication-free solution exists and was found. More generally, if one does not exist, the heuristic will evaluate many solutions and will pick the best one it finds.

5.2.2 Details and fine tuning

An algorithmic outline of the heuristic method is presented in Figure 5.3. Some of the issues in the algorithm are discussed in this section. These are determining how many iterations we should use, solving the problem of local minima, and a heuristic to ensure progress in the algorithm.

Number of iterations The length of the longest path in the bipartite graph seems to be a reasonable bound on the number of iterations. This is because we need partitions in one part to propagate changes to distant parts of the graph. The time needed to do this is bound by the longest acyclic path in the graph, and along this

```
Procedure Do_forward_phase()
    for all d ∈ Data_set do
        if Progress_flag[d] then
            l ← largest loop accessing d which induces changed Data_partition[d]
            Data_partition[d] ← Partition induced by Loop_partition[l]
            Origin[d] ← Access function mapping of Origin[l]
        endif
        Inducing_loop[d] ← l
    endfor
end Procedure

Procedure Do_back_phase()
    for all l ∈ Loop_set do
        d ← Array inducing Loop_partition[l] with minimum cost of accessing all its data
        Loop_partition[l] ← Partition induced by Data_partition[d]
        Origin[l] ← Inverse access function mapping of Origin[d]
        if Inducing_loop[d] ≠ l then
            Progress_flag[d] ← false
        endif
    endfor
end Procedure

Procedure Partition
    Loop_set : set of all loops in the program
    Data_set : set of all data arrays in the program
    Graph_G : Bipartite graph of accesses in Loop_set to Data_set

    Min_partitions ← φ
    Min_cost ← ∞
    for all d ∈ Data_set do
        Progress_flag[d] ← true
    endfor
    for i= 1 to (length of longest path in Graph_G) do
        Do_forward_phase()
        Do_back_phase()
        Cost ← Find total cost of current partition configuration
        if Cost < Min_cost then
            Cost ← Min_cost
            Min_partitions ← Current partition configuration
        endif
        if cost repeated then            /* convergence or oscillation */
            for all d ∈ Data_set do           /* force progress */
                Progress_flag[d] ← true
            endfor
        endif
    endfor
end Procedure
```

Figure 5.3: The heuristic algorithm

length changes are propagated . This bound seems to work well in practice. Further increases in this bound did not provide a better solution in the examples we tried.

Local Minima and Progress We had mentioned earlier that the forward phase of the algorithm chooses the new data partition to be one of the partitions induced by the different loops accessing the array. One possibility is to choose the data partition that minimizes cost. In machines with caches, this is the data partition induced by the largest loop accessing the array. However, we notice that in the back phase cost is already being minimized in selecting loop partitions. Hence we have some leeway in applying another rule here, which will help us solve the problem of local minima. A 'local minimal solution' is one at which all local choices of loop and data partitions minimize cost, but the solution is not globally minimal. When we used the largest loop rule in the forward phase and applied it to several small programs the results were reasonable, but the heuristic failed to find optimum solutions in some cases. The reason was that the heuristic had reached a local minimum solution, and made no further progress.

We solved this problem by adding the following heuristic. We want the algorithm not to remain at a local minima, i.e. we want some rule to ensure progress in the algorithm. So we choose the data partition in the forward phase such that *we prefer to change the partition* if there is a loop which induces a change. However we do not want to change too fast either, before a data partition has had a chance to induce other partitions in the graph to match it. So we add the rider that we will not move in the direction of change if some loop (other than the one which induced this partition in the first place) in the previous iteration had its loop partition induced by this array's current partition.

Another Progress Heuristic Another improvement is that on reaching a situation in which there is convergence or oscillation in iterations, we simply enforce change at all data partition selections in the forward phase. This sets off the heuristic on another path, which sometimes finds a better solution before the iteration count bound is reached. The above two progress optimizations considerably improve the heuristic, and make it more robust. In many small programs we tried, the heuristic now finds the known optimal solution. In the large hydrodynamics application we ran we do not know the optimal, but the heuristic finds a solution with locality quite close to an upper bound we calculated for it.

5.3 Algorithm Complexity

In this section we show that the above heuristic runs in polynomial time in n, the number of loops in the program, and m, the number of distributed arrays they access. We note at this point that an exhaustive search guaranteed to find the optimal for this NP-complete problem is not practical. One algorithm we designed had complexity $O(n^m)$. For the large application we ran, $n = 20$ and $m = 20$, and this would give a time of thousands of years on the fastest machines of today.

Theorem 3 *The time complexity of the above heuristic is* $O(n^2m + m^2n)$.

To find the the complexity of the heuristic, we note that the number of iterations is the length of the longest acyclic path in the bipartite graph, which is bounded above by $n+m$. The time for one iteration is the sum of the times of the forward and back phases. The forward phase does a selection among m possible loop partitions for each of n loops, giving a bound of $O(nm)$. The back phase does a selection among n possible data partitions for each of m arrays, giving a bound of $O(nm)$. Thus overall the time is $O((n+m)mn) = O(n^2m + m^2n)$.

Chapter 6

Cache and Data Locality: Relative Benefits and Tradeoffs

This chapter demonstrates through examples the tradeoffs between optimizing for cache and data locality which the heuristic method makes. We also see how architectural parameters of the target machine affect the choice of optimal partitioning found. The relative benefits of cache and data locality are measured by doing data-locality optimizations alone, and then doing both optimizations. Since this chapter does not add anything to the description of the method, but only to an understanding of the same and of the relative benefits of cache and data locality optimization, a reader looking for just an overview of this thesis could skip over to the next chapter.

Section 6.1 shows how cache locality considerations may be overruled by data locality considerations for one example. Section 6.1.1 shows that cache locality effects ultimately dominate over data locality effects for the same example, as the local access latency is increased to make it closer to remote access latency. Section 6.2 shows an example of how an initial cache optimized solution helps. Section 6.3 presents the effect of the same on *conduct*, a large application.

6.1 Cache Locality may be Overridden

In case the target machine architecture has remote access time significantly larger than local access time, cache locality could often be sacrificed in order to satisfy data locality in case their demands conflict during iterations. The Alewife machine is an example of a machine where this is quite likely. See chapter 7 for Alewife access parameters. These decisions are made during the back phase of the heuristic, when the cost estimator decides which effect is more important. This does not mean that cache optimization in this case gave no benefit, as the choice of the initial solution being one optimized for caches will lead to some benefit over choosing the initial solution to be a default non-cache optimized solution.

Here we present an example of such a case. Consider the code fragment below.

```
Doall (i=0:127, j=0:125)
  A[i,j] = B[i,j]+B[i,j+2]
EndDoall
Doall (i=0:125, j=0:127)
  A[i,j] = B[i,j]+B[i+2,j]
EndDoall
```

The first loop asks for an initial solution based on cache locality as blocked columns, while the second asks for blocked rows. These solutions minimize the cumulative footprint of array B in both loops. Let us consider the portion of the graph with array B only, as that is the only array which is shared or has any demand based on cache locality. Figure 6.1 shows this initial solution.

Figure 6.2 shows the first iteration. In the forward phase, array B gets a data partition based on its largest accessing loop. Here since both loops are of the same size, (say) loop 2 is picked. In the back phase, loop 1 has a choice of either retaining its current partition to optimize cache locality or changing to a blocked row partition to optimize data locality. For Alewife's parameters, our cost estimator found it better

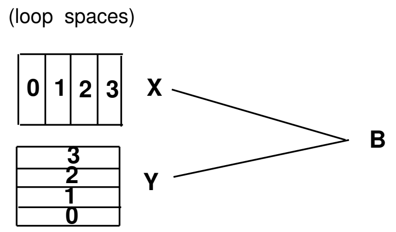


Figure 6.1: Initial solution

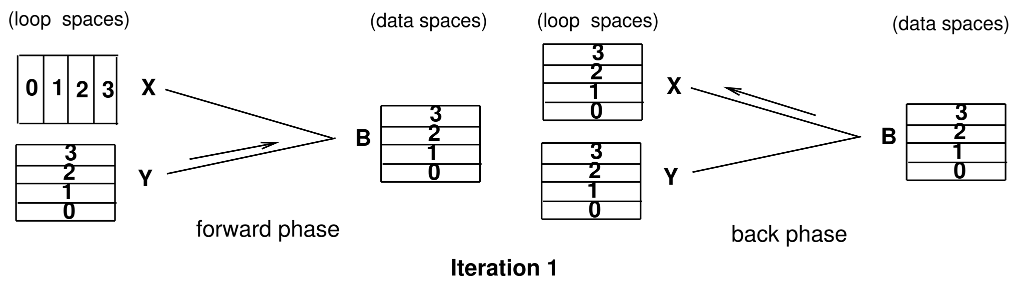


Figure 6.2: Iterations

to satisfy data locality. This choice is shown in the figure, and is the best solution found.

6.1.1 Crossover point

Suppose instead of Alewife's parameters, the cost model was given input parameters such that cache accesses were much cheaper than local accesses, but the latter only slightly cheaper than remote accesses. In such a case it may be possible that cache locality considerations override data locality considerations. To see at what point cache locality dominates we plotted a curve of total estimated access time while varying local access time, keeping cache and remote access times fixed. We fix T_C = 4 cycles, and T_R = 240 cycles. Since remote and local accesses fetch cache lines (assumed 16 bytes), this is an effective remote access time of about 120 cycles for 8 byte floating point numbers (cache:remote = 4:120 = 1:30). We vary T_L between these two values, and the result is shown in figure 6.3.

The crossover point is when, as T_L is increased, the first loop retains its initial

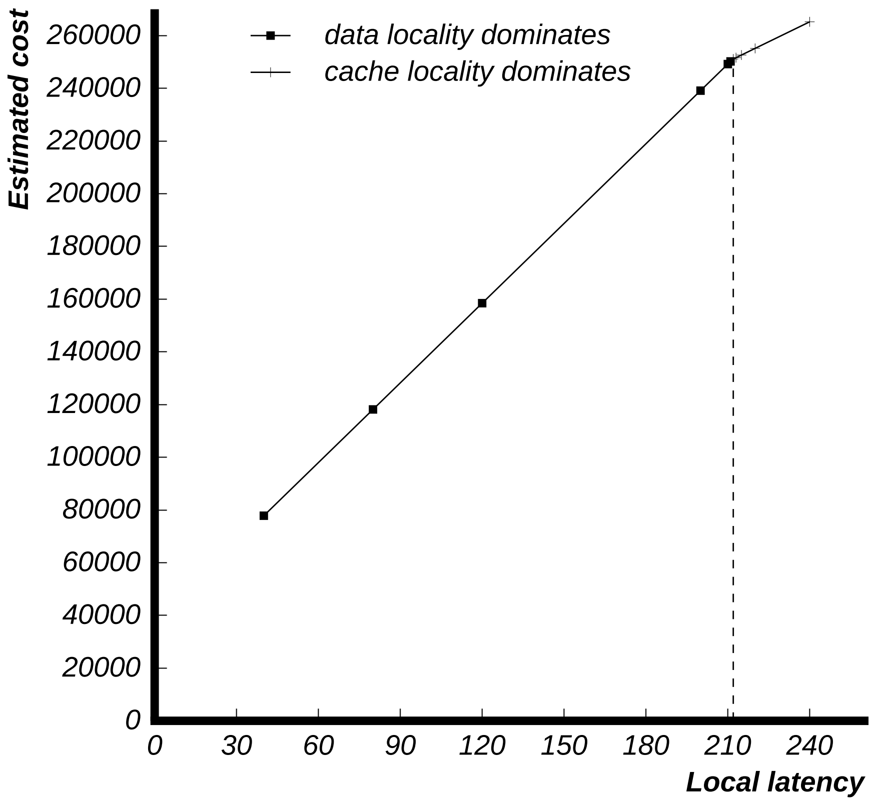


Figure 6.3: Crossover point from data to cache locality domination

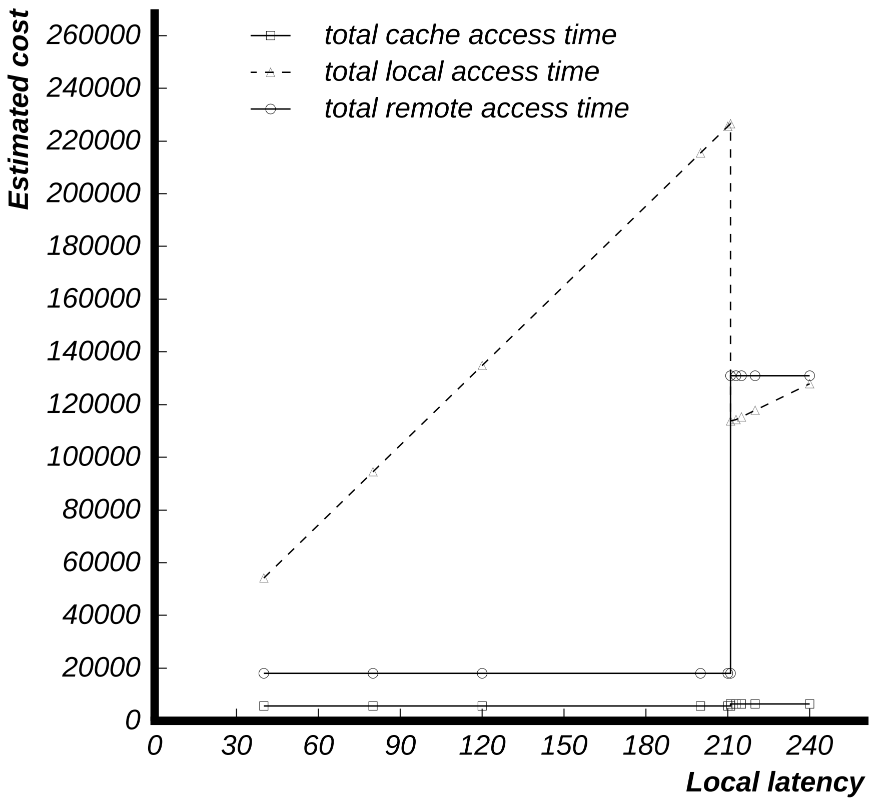


Figure 6.4: Contributions of cache, local and remote accesses to total

cache optimized tile in the back phase of iteration 1. It was measured by observing the actual partitions generated. In this case, the crossover occurred when $T_L = 212$ cycles.It was found that cache locality dominates only when T_L is quite to T_R, and much larger than T_C. For other examples, the crossover point might be lower or higher.

In order to better understand this solution, plots of the total remote, local and cache access time contributions are presented as T_L is varied, in figure 6.4. The sum of the three contribution curves equals the curve in figure 6.3. As the partitioning solutions on each side of the crossover point are the same, but different from the solutions on the other side, the remote and cache contributions remain constant on each side, as T_R and T_C remain fixed. But as T_L is varied, the local contribution increases on both sides. However, we note that the local timing is a larger fraction of the total time on the left hand side of the crossover point. This is expected, as the solution is optimized for data locality on the left hand side, and most data is local.

6.2 Initial cache optimized solution may help

In the above example, we saw a case where cache locality considerations were overridden by data locality considerations .Here we present an example where having the initial solution as cache-optimized lead to a gain, even though it was relatively small. The gain was over a default initial partition where all the loops were partitioned in a square blocked fashion. The globally optimizing heuristic was then run on each of the initial solutions, and the estimated costs compared.

We did this comparison on the program fragment below.

```
Doall (i=0:255, j=0:255)
  A[i,j] = 1
EndDoall
Doall (i=0:251, j=0:255)
```

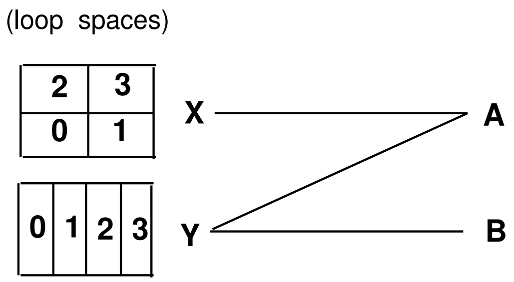


Figure 6.5: Cache-optimized initial solution

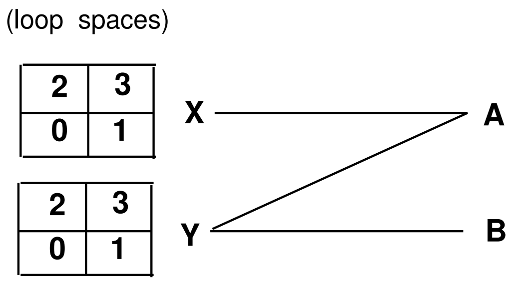


Figure 6.6: Default initial solution

```
  B[i,j] = A[i,j]+A[i,j+4]
```
EndDoall

Figure 6.5 shows the cache-optimized initial solution, while 6.6 shows the default square blocked initial solution. We ran the global heuristic beginning with each of these solutions initially for Alewife's access parameters, and then for a Uniform memory access (UMA) machine ($T_L = T_R$), giving two graphs each of two curves. The curves for Alewife are shown in figure 6.7, while the curves for a UMA machine with $T_L = T_R = 240$ cycles, $T_C = 4$ cycles are shown in figure 6.8.

We see that a benefit for using an initial cache-optimized solution was seen in both types of machines, as expected. The gains were small, but significant (3 to 6%). We also found that no further gains were obtained when the cache effects due to peripheral footprints were taken into account in subsequent iterations. This was due to the effect described in section 6.1 for NUMA machines. For UMA machines, of course, the first iteration itself yields the optimal solution.

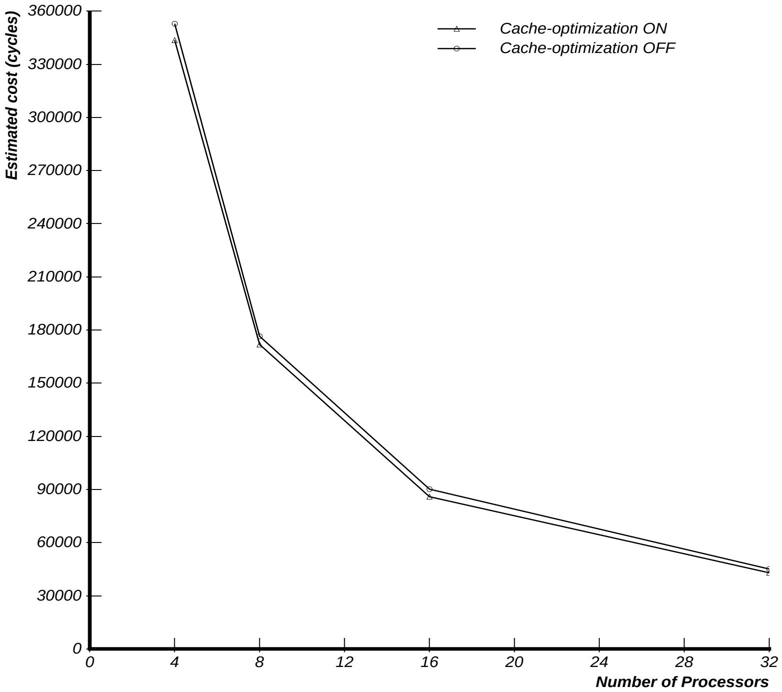


Figure 6.7: Effect of cache optimization on code fragment(Alewife)

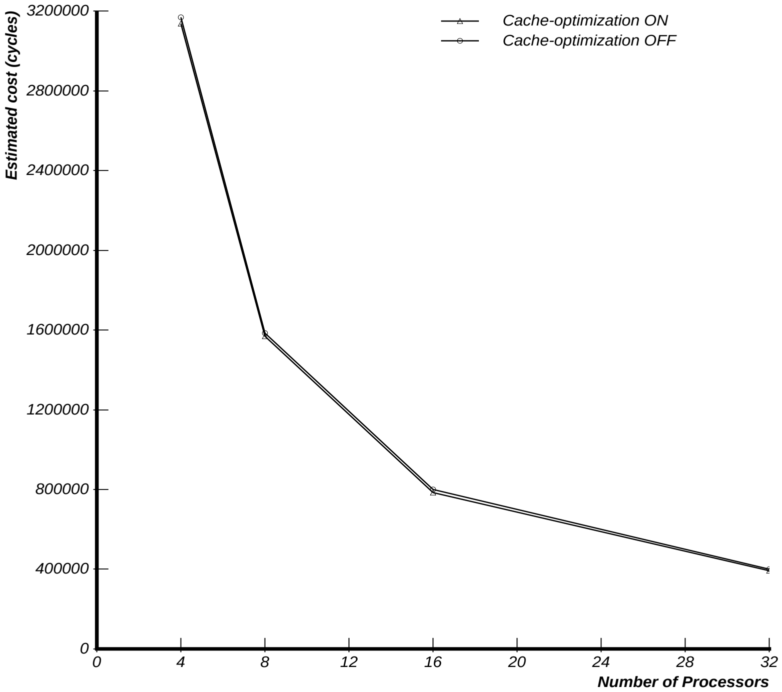


Figure 6.8: Effect of cache optimization on code fragment(UMA machine)

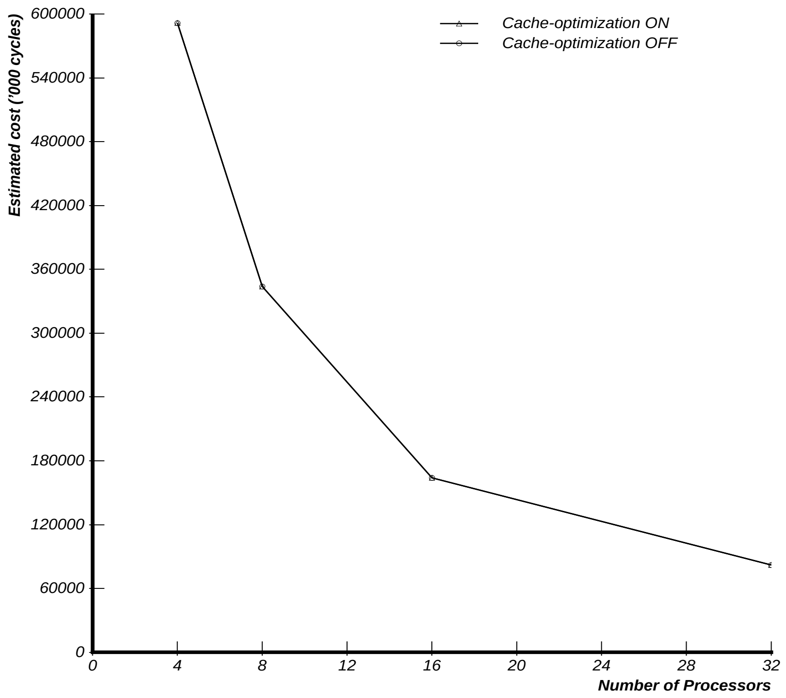


Figure 6.9: Effect of cache optimization on *Conduct*(Alewife)

We conclude that an initial cache optimized solution is a useful optimization for some programs.

6.3 Effect of Cache Optimization on *Conduct*

We performed the same experiment as described in the previous section on the *conduct* routine of the SIMPLE application, a hydrodynamics code from the Lawrence Livermore National Lab, with 20 loops and 20 arrays. That is, estimated costs for a large version of *conduct* were obtained for initial solution cache optimization turned on and off, for both Alewife and a UMA machine. The curves for Alewife are shown in figure 6.9, while the curves for a UMA machine are shown in figure 6.10.

In this case we found measurable but almost negligible benefit in each case. The two curves for both machines were so close that they seem to coincide on the plot (they differed by at most about 0.1%). In the case of Alewife, estimated costs for lower

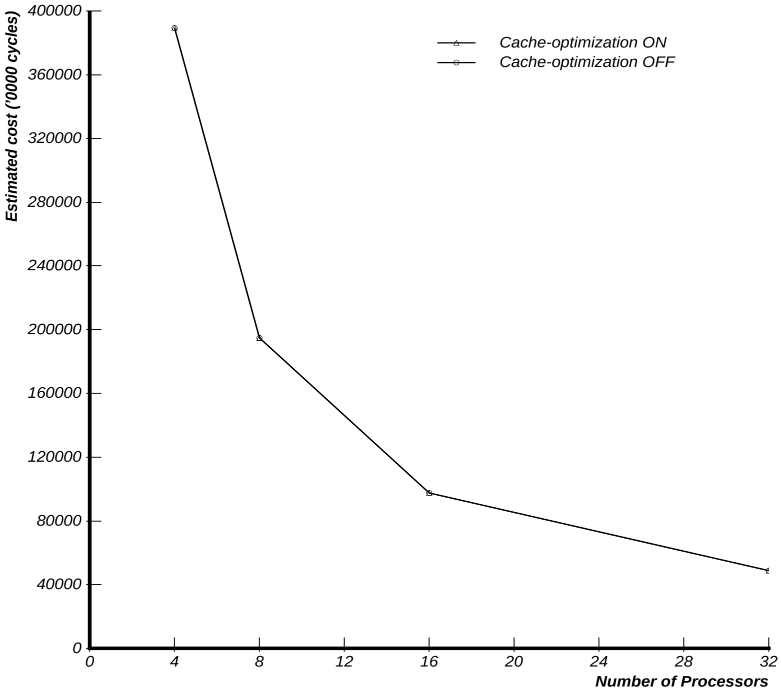


Figure 6.10: Effect of cache optimization on *Conduct*(UMA machine)

number of processors was actually marginally higher with the optimization turned on. This is explained as there is a higher probability of tiles matching up with each other in iterations if they were all partitioned in the same default way to begin with. With higher number of processors, the benefit of cache optimization rises as the peripheral footprint area becomes a larger percentage of a reducing total area per tile, and a net gain in the optimization is seen. In the case of UMA machines, there was always a benefit, but a very small one. In any case the two curves never differ by more than 0.1%, and hence any difference is insignificant.

The reason for the lack of benefit of cache optimization on *conduct* was seen on examination of the partitions generated. It turned out that for the particular example of *conduct*, by coincidence, the cache-optimized initial solutions demanded for the tiles was exactly the same as the default for all the 20 loops save one.

Hence we conclude that an initial cache optimized solution may not make a difference for some programs, and leads to gains in others.

Chapter 7

Results

We have implemented the algorithm described in this thesis as part of our compiler for the Alewife [3] machine. The Alewife machine implements a shared global address space with distributed physical memory and coherent caches. The nodes contain slightly modified SPARC processors and are configured in a 2-dimensional mesh network. The approximate Alewife latencies are: 2 cycle cache hit (4 bytes), 3 cycle cache hit (8 bytes), 11 cycle local memory hit, 40 cycle remote access assuming no contention. Local and remote accesses fetch cache lines (16 bytes). The last number will be larger for large machine configurations.

As an example, we ran the *conduct* routine from the SIMPLE application , a hydrodynamics code from the Lawrence Livermore National Lab, on the Alewife machine simulator, known as NWO [7]. This is the same code used as the example in [4]. It has 20 loop nests. They used a problem size of 1K by 1K but, because we were using a simulator, we used a problem size about 50 times smaller. We use a static partition for the data. Combining the possibility of runtime data migration as in [4] with our algorithm might well improve performance.

We ran the *conduct* code using three different data partitions:

global This is the partition obtained by running the algorithm described in this thesis.

local To get this partition, we do the loop part of the analysis to determine the loop partition, and then partition each array by using the loop partition of the largest loop that accesses the array. This analysis proceeds as in **global** but stops before the first backward step of the iteration.

random This partition just allocates the data across processors with no regard for data locality. However it does loop partitioning optimized for cache locality.

Figure 7.1 shows the speedups obtained for the global, local, and random partitions. We see that the global partitioning heuristic provides a significant increase in performance over local, which itself is significantly faster than using a random data partition.

The results for data-only optimization (initial cache optimized solution not used) are not presented. Meaningful results for that case, not affected by the load imbalance because of the small problem size we are using, which can be obtained by scaling the problem size upwards, will require too much simulation time on our current simulator. In lieu of this data, we did present estimated total access times for a scaled-up *conduct* routine with and without an initial cache optimized solution in section 6.3. Simulating a larger problem size should not be a problem once the Proteus [5] simulator for Alewife is available.

We see that the local version does pretty well. This is partly because the remote latency of the machine we simulated is quite low. Future work will have much more data, including a larger problem size and larger number of processors, as well as results of varying remote memory latency. The different partitions impact the percentage of local versus remote references quite dramatically as shown in Figure 7.2. In all cases the cache hit rate was around 88%.

The overall speedups are not that large compared to those shown in [4] for four reasons.

1. The problem size we used is 50 times smaller resulting in more overhead associated with barriers.

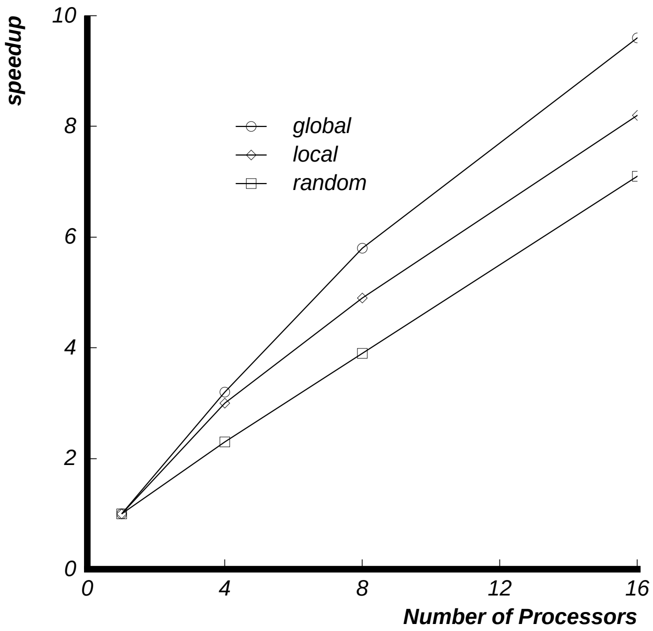


Figure 7.1: Speedup over sequential for conduct. Problem size 153 x 133, double precision.

2. The Stanford Dash [12] machine distributes data which does not fit in local memory across the network in the one-processor run case, which Alewife does not do. Hence their sequential run would be relatively slower than ours, giving a larger speedup for Dash.

3. Our simulation only counts one cycle for each floating-point operation, effectively increasing the communication to computation ratio and lowering speedup.

4. We do not dynamically relocate data when the direction of parallelism changes as they do. This results in many more remote references. In fact, our measurements are similar to what they show when they do not do the dynamic movement.

It still appears that our method significantly decreases the number of remote references even though the overall performance impact depends on these other factors.

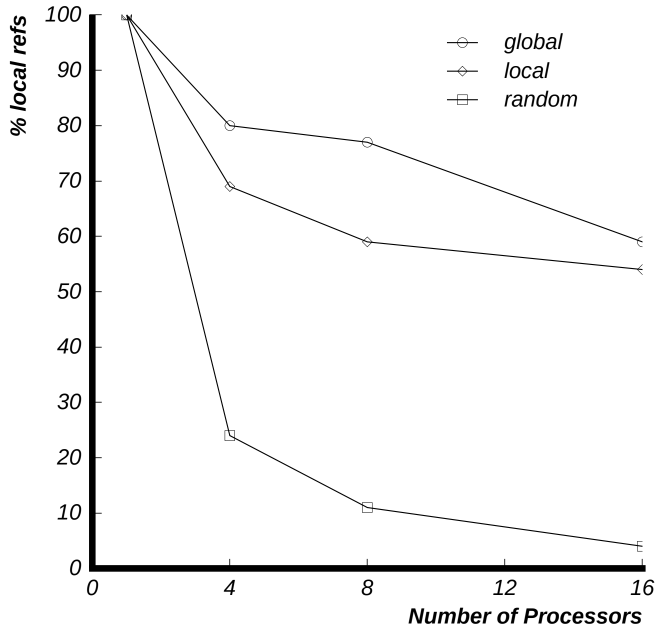


Figure 7.2: Percentage of total array references that were local.

Chapter 8

Conclusions and Summary

This thesis has presented a method and algorithm for automatically partitioning multiple loops and data by evaluating the conflict between cache locality and data locality. By making the approximation that an array reference in a loop partition will have all access local or all remote, we can use a simple cost function to guide a heuristic search through the global space of loop and data partitions. We have implemented this method in our compiler and shown results for a routine of a large application.

Obtained results show that global data partitioning is an important factor in getting good performance. Cache locality optimization is somewhat less important for machines like Alewife, in which remote latency is significantly more expensive than local latency. However the relative importance of doing cache locality optimization increases as local access time becomes a larger fraction of remote access time. In the extreme case when they are equal (UMA machines), data locality holds no meaning, and only cache locality optimization provides gains.

A contribution of this thesis is that this method attempts to do the best possible irrespective of the type of machine.

8.1 Future work

In order to evaluate these techniques fully future work will include runs on larger numbers of processors with more applications. Current results indicate that combining loop and data partitioning is important in obtaining the best performance on cache-coherent distributed memory multiprocessors.

Prefetching is a promising technique for latency reduction in multiprocessors. Software-controlled prefetching [6, 11, 13, 15] involves placing prefetch instructions in the compiler generated code. Future compiler implementations could incorporate prefetching and our partitioning scheme, and measure how well they complement each other.

We would like to add the possibility of copying data at runtime to avoid remote references as in [4]. This factor could be added to our cost model. We do however suspect that data relocation is probably less important in cache-coherent shared memory machines than in message passing machines.

Bibliography

[1] S. G. Abraham and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.

[2] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd International Conference on Parallel Processing*, St. Charles, IL, August 1993. IEEE. A version of this paper appears as MIT/LCS TM-481, December 1992.

[3] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[4] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of SIGPLAN '93, Conference on Programming Languages Design and Implementation*, June 1993.

[5] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[6] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 40–52. ACM, April 1991.

[7] David Chaiken. NWO User's Manual. ALEWIFE Memo No. 36, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1993.

[8] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[9] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[10] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[11] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th International Conference on Computer Architecture*, Toronoto, May 1991.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[13] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[14] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[15] Monica S. Lam Todd C. Mowry and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth ACM Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 1992.

[16] C.-W. Tseng. *An Optimizing Fortran D compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, Jan 1993. Published as Rice COMP TR93-199.

[17] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 91 Conference Programming Language Design and Implementation*, pages 30–44, 1991.

[18] Michael E. Wolf and Monica S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. In *The Third Workshop on Programming Languages and Compilers for Parallel Computing*, August 1990. Irvine, CA.