LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-466

HIERARCHICAL COMPILATION OF MACRO DATAFLOW GRAPHS FOR MULTIPROCESSORS WITH LOCAL MEMORY

G. N. Srinivasa Prasanna
Anant Agarwal
Bruce R. Musicus

October 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Hierarchical Compilation of Macro Dataflow Graphs for Multiprocessors with Local Memory [1]

G. N. Srinivasa Prasanna, Anant Agarwal, and Bruce R. Musicus
Laboratory for Computer Science, Rm. 624B,
Massachusetts Institute of Technology,
Cambridge, MA 02139
and
Bolt, Beranek, and Newman, Inc.
prasanna@masala.lcs.mit.edu
agarwal@mit.edu
musicus@bbn.com

February 14, 1992

Abstract

This paper presents a hierarchical approach for compiling macro dataflow graphs for multiprocessors with local memory. Macro dataflow graphs comprise several nodes (or macro operations) that must be executed subject to prespecified precedence constraints. Programs consisting of multiple nested loops, where the precedence constraints between the loops are known, can be viewed as macro dataflow graphs. The hierarchical compilation approach comprises a processor allocation phase followed by a partitioning phase. In the processor allocation phase, using estimated speedup functions for the macro nodes, computationally efficient techniques establish the sequencing and parallelism of macro operations for close-to-optimal run times. The second phase partitions the computations in each macro node to maximize communication locality for the level of parallelism determined by the processor allocation phase. The same approach can also be used for programs consisting of multiple loop nests, when each of the nested loops can be characterized by a speedup function.

These ideas have been implemented in a prototype structure-driven compiler, SDC, for expressions of matrix operations. The paper presents the performance of the compiler for several matrix expressions on a simulator of the Alewife multiprocessor.

Keywords: Parallel compilation, cache-coherent multiprocessors, distributed-memory multiprocessors, communication locality, task scheduling, parallel processing, parallelizing compilers.

[1] A short paper on the processor allocation phase of SDC has been submitted to ICPP'92.

1 Introduction

Multiprocessors rely on careful allocation of their processing, communication, and memory resources to computations for achieving high performance. While it is possible for programmers to carefully orchestrate their computations, producing correct and efficient programs is extremely difficult. The problem is even more severe on multiprocessors with complex memory hierarchies. For many classes of problems, which display a known structure, or which are amenable to static analysis, it is possible for a compiler to derive programs that exhibit close-to-optimal run times.

A compiler must address several important inter-related issues. First, the workload of the program must be equitably distributed across all the available processors, keeping the communication between tasks to a minimum. This is called the *partitioning* problem. Next, the set of tasks for each processor must be sequenced to satisfy all precedence constraints, while minimizing processor idle time. This is the *scheduling* problem. Both partitioning and scheduling are difficult, NP-Hard problems.

However, optimal compilation can be greatly simplified if the computation has a hierarchical structure, that is, if it can be represented as a *macro dataflow graph*. A macro dataflow graph is composed of macro nodes, where each macro node has internal structure. For example, a matrix expression can be represented as a macro dataflow graph, where each node corresponds to basic matrix operators, such as matrix adds and multiplies. Programs consisting of multiple nested loops can often be represented as macro dataflow graphs, where each node in the dataflow graph corresponds to a nested loop, and where some form of synchronization (e.g., counting semaphores) among the processors executing the nested loop establishes the precedence constraints between different loop nests.

We simplify the compilation of macro dataflow graphs by compiling separately the two levels of hierarchy. In the first phase, called *processor allocation*, the complete macro dataflow graph is compiled, treating each macro node as a unit. In this step, the sequencing of macro nodes, or *scheduling*, and the number of processors assigned to each macro node, or *node parallelism*, is determined. This step uses the speedup functions of each macro node. After the processor allocation phase, the computations within each macro node are *partitioned* among the processors assigned to that node for communication efficiency. This divide and conquer strategy not only reduces the combinatorial complexity of compiling, but affords further simplifications if we exploit our knowledge of the structure of computation within each macro node.

Our work in processor allocation complements recent work [1, 2, 3] in partitioning nested loops. Since nested loops can be treated as macro nodes, our work is equivalent to determining the optimal schedule and loop parallelism for a program with *multiple interdependent loop nests*. Their work in partitioning loop nests treated each loop nest separately, and assumed a certain number of processors over which to partition the loop. The techniques discussed in this paper can be used to choose the number of processors to be assigned to each loop nest. Our work can also be used in conjunction with the techniques to compile possibly imperfectly nested loops developed by Polychronopoulos et al. [4] and Wang et al., [5]. Their algorithms did not attempt to run disjoint, independent subloops at a given nesting level in parallel as does our scheme when the precedences between different portions of the loop nest are known. Our techniques

can be used to yield significant performance gains by exploiting this interloop parallelism.

In this paper, we describe computationally efficient techniques for processor allocation in macro dataflow graphs. We develop algorithms for determining node parallelism and sequencing, and describe the design of a structure-driven compiler (SDC) using these algorithms for matrix expressions. We also discuss techniques for partitioning the macro nodes corresponding to matrix operations, since these techniques provide the speedup functions necessary for processor allocation. Our partitioning methods use geometric bin packing techniques, which allow us to take into account boundary truncation effects and mismatched numbers of processors.

Matrix expressions were chosen because many important examples have macro dataflow graphs which exhibit simple data-independent control, and have a wide variety of graph structures well suited to automatic compilation. The dataflow graphs of the macro nodes (matrix operators) are regular and well characterized, enabling speedup functions to be derived by simple analysis. Also, many algorithms in numeric computation implicitly require computation of matrix expressions (e.g., computation of an entire array in a nested loop).

Some examples of matrix expressions are shown below, where all operators are matrix operators. In all that follows, the terms "macro node," "macro operator," and "matrix operator" mean the same.

$$
\begin{aligned}
Y &= A(B + CD) && \text{– Simple Matrix Expression} \\
Y &= a_0 + a_1 A + a_2 A^2 + a_3 A^3 + \cdots + a_N A^N && \text{– Matrix Polynomial} \\
Y &= WX && \text{– Fourier Transform, where matrix W: } w_{kl} = e^{-j\frac{2\pi kl}{N}}
\end{aligned}
$$

We present experimental results for several matrix expressions compiled using SDC on a simulator of the Alewife multiprocessor. Alewife [6] is a distributed shared-memory multiprocessor being developed at MIT. Our experiments indicate that careful processor allocation yields speedups that far exceeds the speedups due to techniques that allocate all the processing resources to each nested loop in turn.

1.1 Hierarchical Compilation

In principle, general purpose approximation algorithms for partitioning and scheduling can be applied to a completely expanded dataflow graph, where the internals of each macro node are completely exposed. These resultant partitions and schedules are close to being globally optimal. However, when data sets have $\Theta(N)$ computations, to be compiled on P processors, these general techniques remain computationally feasible only for very small N, for they exhibit average compilation times $\Theta(N)$ to $\Theta(N^3)$.

When the computational graphs display hierarchical structure (e.g., matrix operations manifested in a program as nested loops), a hierarchical compilation strategy simplifies partitioning and scheduling of the complete computation by performing compilation in two steps.

First, the processor allocation step determines both the optimal number of processors computing every macro node (node parallelism), as well as the order in which macro nodes are

computed (sequencing). This step uses the *speedup functions* of the macro nodes derived by analyzing the internal dataflow graphs of the macro node. Given the macro dataflow graph representation of a set of interdependent nested loops and the speedup functions of each loop nest, the same procedure can determine the optimal number of processors assigned to each loop nest, and the sequencing of the loops. Processor allocation is a generalization of classical scheduling and is therefore sometimes referred to as *generalized scheduling.*

Next, the dataflow graph of the individual macro nodes is independently partitioned and scheduled, for the parallelism determined above. The partitioning is done so that communication incurred in computing the macro node is minimized, while maintaining an even load on each processor. We exploit the *regular structure* of the dataflow graphs of these operators and use bin packing techniques to achieve close-to-optimal partitions.

Our work in partitioning differs from the work of others [1, 2, 3] not only in the techniques used for partitioning, but in our emphasis on partitioning for an arbitrary number of processors, which addresses fragmentation at the boundaries of the dataflow graph when the number of processors is mismatched to the problem size. We note, however, that although our techniques yield close-to-optimal partitions for structured graphs, they are less general than previously reported methods.

This paper has two major parts, corresponding to each step of the hierarchical compilation strategy. First, Section 3 presents processor-allocation techniques to determine the parallelism and sequencing of all the operators in a matrix expression. Next, Section 4 presents techniques for partitioning the dataflow graphs of the matrix operators, for the given parallelism.

1.2 A Simple Example

Consider the following matrix expression (denoted $g1$),

$$
\begin{array}{l}
(+\ (\times\ A_0\ A_1) \\
\quad (+\ (\times\ (+\ A_2\ A_3) \\
\qquad\quad A_4) \\
\qquad A_5))
\end{array}
$$

whose macro dataflow graph appears in Figure 1(a). The macro dataflow graph is a tree with two roughly equal sized branches, one with a single multiply, and the other with a multiply and two additions. This matrix expression is equivalent to the set of nested loops shown in Figure 2, where the usage pattern of the matrices establishes the precedence constraints. Assume we want to compile this on five processors.

Figure 1(b) illustrates a general purpose compilation algorithm, which expands the dataflow graph of all the five operators (partially or completely) to yield a large, flattened dataflow graph for the expression. Then it partitions and schedules the resulting graph, ignoring pre-existing structure within each operator, yielding five threads of computation. Two processors cooperate to compute the smaller branch of the tree (single multiply), while the remaining three compute the branch of the tree consisting of the two adds and a multiply. Although this resulting partition and schedule is globally optimal, this strategy is time consuming.

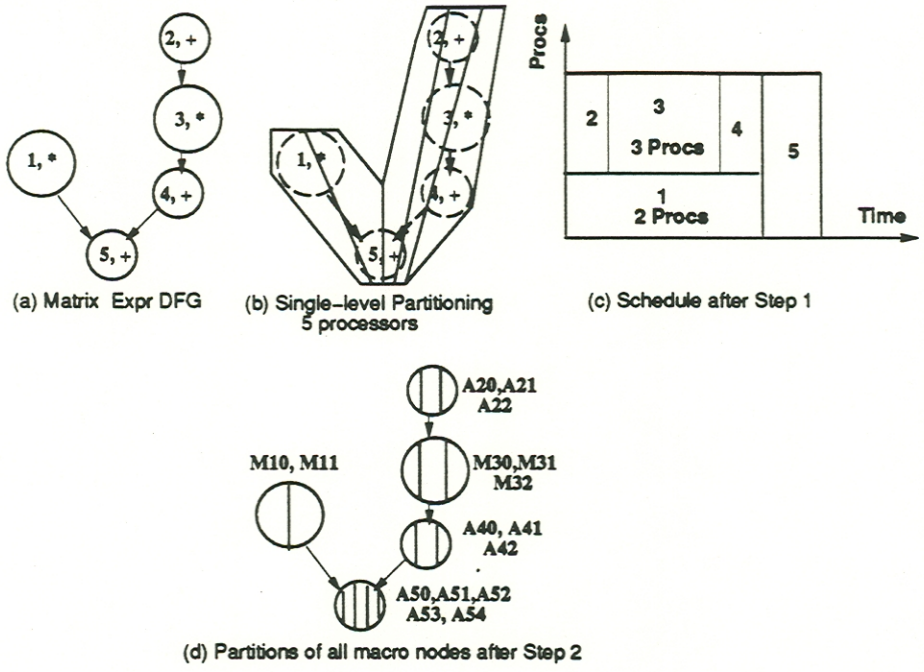

Figure 1: The hierarchical compilation paradigm.

Figure 1(c) and (d) illustrate the hierarchical compilation strategy. First, the sequencing and parallelism of all the five matrix operators is determined using the algorithms to be presented in Section 3. One of these (the Tree algorithm) makes the following choices for the sequencing and parallelism. The two branches of the tree are started simultaneously, and the processors are distributed among the two branches so that they also finish at the same time. Two processors are assigned to the smaller branch, while three processors are assigned to each operator in the other branch. Finally, the last addition is run on all five processors. At this point, the sequencing and parallelism of every operator has been determined; the Gantt chart in Figure 1(c) depicts the resulting schedule and processor allocation.

Next, the optimal partitioning algorithms (Section 4) are used to partition each operator for the number of processors determined above. As depicted in Figure 1(d), the computations in the matrix multiply (operator 1) is partitioned into two chunks, M_{10} and M_{11}, for execution on two processors to balance their load and minimize communication.

Multiprocessor code for the expression is now generated by spawning five threads, with each thread computing a set of the chunks comprising the partitioned dataflow graph, as shown in Figure 1.2. Synchronization points are inserted to ensure completion of computation of an operator before computation on successors begin.

The hierarchical partition and schedule is similar to the globally optimal schedule. Exploiting the hierarchy results in major simplifications in compilation, since the scheduler deals with just five macro nodes. If the matrix sizes are $\Theta(N)$, a general purpose algorithm would have to handle $\Theta(N)$ simple nodes, exhibiting runtimes $\Theta(N)$ to $\Theta(N^3)$.

```
 /* Input: NxN Matrices A0, A1, A2, A3, A4, A5.
  * Output: Matrix T4
  */

 /* Multiply 1 */
doall i = 1 to N
  doall k = 1 to N
    {
      T0[i,k] = 0;
      do j = 1 to N
        T0[i,k] = T0[i,k] + A0[i,j] * A1[j,k];
    }

 /* Addition 2 */
doall i = 1 to N
  doall j = 1 to N
    T1[i,j] = A2[i,j] + A3[i,j];

 /* Multiply 3 */
doall i = 1 to N
  doall k = 1 to N
    {
      T2[i,k] = 0;
      do j = 1 to N
        T2[i,k] = T2[i,k] + T1[i,j] * A4[j,k];
    }

 /* Addition 4 */
doall i = 1 to N
  doall j = 1 to N
    T3[i,j] = T2[i,j] + A5[i,j];

 /* Addition 5 */
doall i = 1 to N
  doall j = 1 to N
    T4[i,j] = T0[i,j] + T3[i,j];
```

Figure 2: A set of loop nests equivalent to expression $g1$.

The rest of the paper sketches these ideas in detail. Section 2 describes the algorithmic and architectural simplifications required to make the problem tractable. Section 3 sketches processor-allocation techniques used to determine the sequencing and parallelism of all macro nodes (matrix operators). Section 4 describes the techniques used to partition and schedule individual matrix operators, given the node (operator) parallelism. Section 5 provides details of our implementation, and Section 6 presents a detailed illustration of the compiler's operation. Section 7 presents experimental results. Section 8 summarizes related work and Section 9 concludes the paper.

```
Spawn threads T0, T1, T2, T3, T4.
Thread T0:
  Compute M10; Wait for M11, A40, A41, A42 to complete; Compute A50;
Thread T1:
  Compute M11; Wait for M10, A40, A41, A42 to complete; Compute A51;
Thread T2:
  Compute A20; Wait for A21, A22; Compute M30; Wait for M31, M32;
     Compute A40; Wait for M00, M01, A41, A42; Compute A52;
Thread T3:
  Compute A21; Wait for A20, A22; Compute M31; Wait for M30, M32;
     Compute A41; Wait for M00, M01, A40, A42; Compute A53;
Thread T4:
  Compute A22; Wait for A20, A21; Compute M32; Wait for M30, M31;
     Compute A42; Wait for M00, M01, A40, A41; Compute A54;
```

Figure 3: Psuedo code produced by hierarchical compilation.

2 Simplifying Assumptions

We make several simplifying assumptions in the architectural model and in the scheduling algorithms to make the problem tractable.

2.1 Partitioning and Scheduling

The globally optimal partition and schedule requires handling the complete dataflow graph as a unit. Communication between macro nodes as well as that within a macro nodes influences the result. Furthermore, the optimal partition and schedule may have portions of a macro node being computed, before predecessors of other portions of the same macro node have finished (non-strict execution).

SDC makes the following simplifying assumptions. In the first step, it determines the number of processors assigned to a macro node using the speedup functions of the macro nodes, treating each macro node independently as a unit. Therefore, the schedules are necessarily strict – all predecessors of a macro node are fully computed before it can start execution. We assume that the speedup functions can either be predicted or empirically determined. This assumption is true for most operators found in matrix arithmetic (and also many nested loops).

In the second step, each macro node is partitioned independently for the number of processors determined in the first step. We ignore communication between the macro nodes. This assumption is reasonable if communication within a macro node dominates the communication between macro nodes.

2.2 Architectural Abstraction

The hierarchical compilation strategy exploits compile-time knowledge of the multiprocessor architecture to estimate various quantities, for example, speedup functions, and necessitates

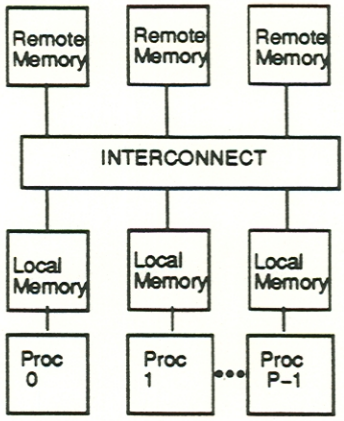

Figure 4: Multiprocessor model.

Operation	Time
Add	T_a
Multiply	T_m
Single Word Remote Access	T_u
Single Word Fetch and Add	T_{fa}

Table 1: Multiprocessor parameters.

a simple characterization of the multiprocessor architecture. Our architectural abstraction, depicted in Figure 4, models a distributed-memory multiprocessor with P processors. Each processor has associated fast local memory (or a cache), and accesses global memory and other processors through an interconnection network. Global memory is distributed among several remote memory modules, as depicted in the figure. During the computation of a macro node, we assume that shared data required for the computations are fetched into the fast local memory from global memory, and that the result of the computations are stored in global memory.

Table 1 lists important architectural parameters. The processor is parameterized by its operation times for additions, T_a, and operation times for multiplications, T_m. These operation times include the times needed to access locally available data (in cache or fast local memory).

We assume that a single-word access from remote memory takes time T_u. We further assume that P such accesses, one from each processor, and each to a different datum, can occur simultaneously. The fixed remote access cost assumes that all remote memories are equidistant from each processor, and that remote data access times are independent of the location of the datum in the multiprocessor system.

The basic synchronization operation is a *fetch-and-add* on a shared datum [7]. The fetch-and-add operation allows an atomic update of a global datum. For matrix multiplies, for example, the fetch-and-add operation allows synchronized accumulates to compute each element of the resulting product matrix. The fetch-and-add operation can also be used in a software combining tree to implement distributed semaphores [8], which are required to enforce the precedence constraints in the macro dataflow graph. Let T_{fa} denote the time required for

a fetch-and-add on a remote datum, (excluding the addition cost T_a, and assuming limited contention). As for remote accesses, we assume P fetch-and-adds, each on a distinct datum, can take place simultaneously.

In the machine used for the experimental measurements, Alewife, the cost of a fetch-and-add is roughly twice the cost of a remote memory access ($T_{fa} \approx 2T_u$), because of contention and because of the higher likelihood of requiring invalidations to other caches. Alewife implements the fetch-and-add on remote data by fetching the data into the local cache and performing a local add. More details about the architecture are in Section 5.1.

3 Determining Sequencing and Parallelism of Macro Nodes

The processor allocation problem comprises two tasks: the number of processors computing every macro node (node parallelism) has to be computed, and the sequencing of the macro nodes has to be determined. Since processor allocation determines macro node parallelism in addition to sequencing, it is a generalization of classical scheduling, and will also be referred to as *generalized scheduling* [9]. We have developed techniques based on *optimal control theory* for this purpose. The macro dataflow graph representation of loop nests allows these techniques to be used for optimally compiling a set of interdependent nested loops.

We first present a simple intuitive characterization of the processor allocation problem. Then we present a formulation based on optimal control theory and summarize the results that emerge. We then describe an optimal processor allocation technique for tree structured macro dataflow graphs, and a greedy processor allocation heuristic for other graphs. The section concludes by discussing how the theoretical results are used to derive practical processor allocation heuristics.

3.1 Intuition

The intuition underlying our algorithms is that as we increase the number of processors allocated to an macro node, overhead of various kinds - scheduling, communication, synchronization – increases. Thereby, the incremental speedup obtained keeps falling, which implies that the speedup functions of the macro nodes are *convex*. Hence overall computation speed is maximized by running concurrently as many macro nodes as the available parallelism allows, using few processors for each macro node. In contrast, running the macro nodes one by one, using all the processors for each macro node, is much slower. Essentially, running many macro nodes in parallel maximizes the granularity of the threads produced from each macro node, thus minimising overhead. This intuition can be given a rigorous foundation using optimal control theory.

3.2 Control Theoretic Formulation of Generalized Scheduling

The fundamental paradigm is to view macro nodes as dynamic systems, whose state represents the amount of computation completed at any point of time. The macro dataflow graph is then viewed as a composite macro node system - the individual macro nodes being its subsystems.

At each instant, state changes can be brought about by assigning (possibly time varying) processing power to the macro nodes. Computing the composite system of macro nodes is equivalent to traversing a trajectory of the macro node system from the initial (all zero) uncomputed state to the final fully computed state, satisfying constraints on precedence and total processing power available. The processors have to be allocated to the macro nodes in such a way that the computation is finished in the minimum time.

This is a classical optimal control problem. The macro node system has to be controlled to traverse the trajectory from start to finish. The resources available to achieve this control are the processors. A valid control strategy never uses more processors than available, and ensures that no macro node is started before its predecessors are completed. A minimal time generalized schedule is equivalent to a time-optimal control strategy (optimal processor-allocation), and this can be formalized as given below.

3.3 Formal Specification

Let $\Omega = \{1, \ldots, N\}$ be a set of N macro nodes to be executed on a system with P processors. Let macro node i have length L_i. That is, L_i denotes the execution time of the macro node on a single processor. A set of precedence constraints is specified, wherein macro node i cannot start until after all its predecessors have finished.

It is convenient to define the *state* $x_i(t)$ of macro node i at time t to be the amount of work done so far on the macro node, $0 \leq x_i(t) \leq L_i$. Let t_i be the earliest time at which all predecessors of i (if any) have finished, so that i can begin running. Thus $x_i(t) = 0$ for $t < t_i$, and $x_j(t_i) = L_j$ for all of i's predecessor macro nodes j. If macro node i has no predecessors, $t_i = 0$.

Let $p_i(t)$ be the processing power (number of processors or processor assignment) applied to macro node i at time t, and let P be the total processing power available. The $p_i(t)$ are all non-negative, and must sum to at most P. Note that we have allowed the $p_i(t)$ to be arbitrary time varying functions, thus allowing arbitrary preemptive (generalized) schedules.

Finally, assume that once an macro node's predecessors have finished, the rate at which it proceeds, $dx_i(t)/dt$, depends in some nonlinear fashion on the amount of processor power applied, $p_i(t)$, but not on the state $x_i(t)$ of the macro node, nor explicitly on the time t. We call this the assumption of *space-time invariant dynamics*. Thus we can write:

$$\frac{dx_i(t)}{dt} = \begin{cases} 0 & \text{for } t < t_i \\ s_i(p_i(t)) & \text{for } t \geq t_i \end{cases} \tag{1}$$

where $s_i(p_i(t))$ will be called the *speedup function*. With no processing power applied, the macro node state should not change, $s_i(0) = 0$. With processing power applied, the macro node should proceed at some non-zero rate, $s_i(p) > 0$ for $p > 0$. We further assume that $s_i(p)$ is non-decreasing, so that adding more processors can only make the macro node run faster. In most of our theory, $s_i(p)$ is taken to be *convex* in p. This convexity reflects the increasing amount of communication, synchronization, and scheduling overhead as the number of processors working on one macro node increases.

Our assumptions about macro node speedup are a simple theoretical abstraction. In effect, this form of the speedup function implies that macro nodes can be dynamically configured into arbitrary numbers of parallel modules for execution on separate processors. Processors can be added or removed at any time, and in such a manner that the processors assigned to the macro node can all do useful work. The speedup depends only on the total number of processors allocated to the macro node at a given time, and is independent of the state or the time variable. Our goal is to finish all macro nodes in the minimum amount of time t^F, by properly allocating processor resources $p_i(t)$.

3.4 Results from Control Theory

The results of time-optimal control theory [9] can be invoked to yield insights into generalized scheduling. The results include:

- General theorems regarding optimal macro node starting and finishing times. One theorem states that a set of independent macro nodes should start and finish together, and be computed simultaneously.
- General rules for simplifying the scheduling problem in special cases. Equivalence of the generalized scheduling problem to constrained shortest path and network flow problems in such cases.
- General purpose heuristics for scheduling, based on the speedup functions of the macro nodes. These techniques are provably optimal in special cases. In particular, a very simple divide and conquer heuristic for tree-structured macro dataflow graphs emerges, which can be shown to be *optimal* for certain types of macro node speedups (e.g., for speedups of the form p^α [9]). The Tree Heuristic is discussed further below.

3.5 Tree Heuristic

The scheduling is especially simple when all speedup functions are of the form $s(p) = p^\alpha$, for the same α. In this case the optimal processor assignment $p_i(t)$ for any macro node i does not vary during its computation, but is constant. This processor assignment will be denoted by p_i itself for simplicity. Moreover, the following graph reduction techniques are available to simplify the scheduling.

A set of macro nodes, $1, 2, \cdots K$, in series can be replaced by an equivalent single macro node (denoted $1:K$) of length, $L_{1:K}$, equal to the sum of the individual macro node lengths, L_i. That is,

$$L_{1:K} = \sum_{i=1}^{K} L_i$$

An optimal generalized schedule (processor allocation), S_R, for the reduced graph maps directly into an optimal generalized schedule for the original graph, S_O, as follows. The processor assignments of each macro node in S_O is equal to that of the composite macro node $1:K$ in S_R, that is

$$p_i[\text{in } S_O] = p_{1:K}[\text{in } S_R] \qquad i = 1 \cdots K$$

Therefore, all macro nodes in a series set have the same processor assignment. A macro node starts as soon as its (sole) predecessor in the series set finishes. S_R, for the reduced graph maps directly into an optimal generalized schedule for the original graph, S_O, as follows. The processor assignments of each macro node in S_O is equal to that of the composite macro node $1:K$ in S_R, that is

$$p_i[\text{in } S_O] = p_{1:K}[\text{in } S_R] \qquad i = 1 \cdots K$$

Therefore, all macro nodes in a series set have the same processor assignment. A macro node starts as soon as its (sole) predecessor in the series set finishes.

A set of parallel macro nodes $1, 2, \cdots K$, can be reduced to an A set of parallel macro nodes $1, 2, \cdots K$, can be reduced to an equivalent single macro node $1:K$ of length, $L_{1:K}$, equal to the $\ell_{1/\alpha}$ norm of the individual macro node lengths, L_i. In other words,

$$L_{1:K} = \ell_{1/\alpha}(L_1, \ldots, L_K) \equiv \left(\sum_{i=1}^{K} L_i^{1/\alpha} \right)^{\alpha}$$

An optimal generalized schedule, S_R for the reduced graph maps directly into an optimal generalized schedule for the original, S_O as follows. All macro nodes in the parallel set are started and finished at the same time. This can easily be shown to imply that the processor assignments among the parallel macro nodes is in proportion to the $1/\alpha$ power of their individual lengths, that is,

$$p_i[\text{in } S_O] = \frac{L_i^{1/\alpha}}{\sum_{j=1}^{K} L_j^{1/\alpha}} p_{1:K}[\text{in } S_R] \qquad i = 1 \cdots K$$

Since trees are recursive series-parallel graphs, these two reductions yield a simple, optimal scheduling technique for trees. Specifically, the tree scheduling algorithm has two steps:

1. We first recursively reduce the entire graph to a single task with p^α speedup, using series-parallel reductions as described above, determining the length (workload) of the tree and all subtrees. Now, the run time of the tree is easily computed.

2. Next, undoing the recursion, we allocate the processing power to each of the series and parallel components according to their lengths, and determine their start and stop times. Continuing recursively, we eventually derive the optimal processor allocation(or generalized schedule) for every task in the original graph.

Pseudo code for the Tree Heuristic is shown in Figure 5. Each node in the data structure contains the length of the corresponding macro node (op-len), the total length of the macro node plus the equivalent length of all its subtrees (tree-len), the processor assignment (proc) to this node, and the start and stop times for the macro node. First, *find_length* is recursively called to determine the lengths of all subtrees. Then *tree_scheduler* uses the above determined lengths to compute the processor assignments and the start and finish times for all nodes. Notice that it attempts to equalize finish times of all predecessors (subtrees) at each node, by properly splitting the processor resource among them.

This Tree scheduling technique is optimal only when all speedup functions are of the form p^α, for the same α. Although the matrix product speedup in Equation 4 (Section 4.3), is not of

```
        struct tree
{
          op_len;       /* The length of the corresponding macro-node */
          tree_len;     /* The length of the corresponding macro-node,
                         * plus lengths of all its subtrees*/
          proc;         /* Processor assignment for this macro-node and
                         * all its subtrees */
          op_start;     /* Start time for this macro-node */
          op_stop;      /* Stop time for this macro-node */
          left_child,
          right_child; /* pointer to children, NIL if a child is absent */
        }

        parallel_length(a, b)
{
   return(1/alpha norm of a,b);
}

        find_length(tree)
        {
           left_len = right_len = 0;
           if (tree.left_child) left_len = find_length(tree.left_child);
           if (tree.right_child) right_len = find_length(tree.right_child);
           tree.tree_len = parallel_length(left_len,right_len) + tree.op_len;
           return(tree.tree_len);
        }

        ;Assumes that find_length has been called already
        tree_scheduler(tree, nproc)
        {
           left_len = right_len =0;
           op_start = 0;
           if (tree.left_child) left_len = tree.left_child.tree_len;
           if (tree.right_child) right_len = tree.right_child.tree_len;
           if (tree.left_child)
           {
              left_proc = nproc * left_len / (left_len + right_len);
              tree_scheduler(tree.left_child,left_proc);
              op_start = tree.left_child.op_stop;
           }
           if (tree.right_child)
           {
              right_proc = nproc * right_len / (left_len + right_len);
              tree_scheduler(tree.right_child,right_proc);
              op_start = tree.right_child.op_stop;
           }
           tree.proc = nproc;
           tree.op_start =op_start;
           tree.op_stop = op_start + tree.op_len / nproc^alpha;
         }
```

Figure 5: Pseudo code for Tree Heuristic.

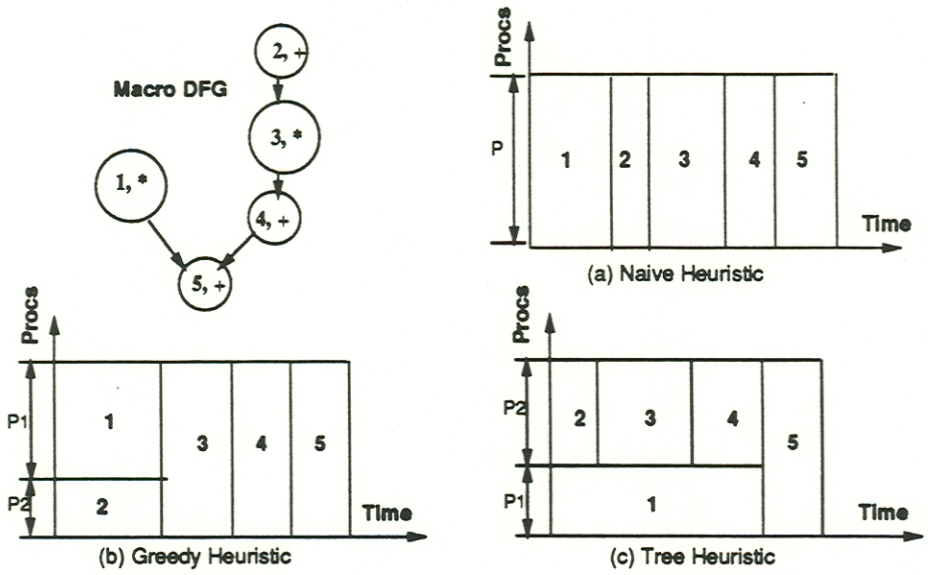

Figure 6: Processor allocation (or generalized scheduling) heuristics.

this form, we can employ this technique as a heuristic, since the speedup for matrix multiply can be approximated as p^α, and an appropriate α can be empirically determined (see Section 7.1). We show that the technique is robust for the value of α used.

3.6 Other Processor Allocation Heuristics

For purposes of comparison, we have incorporated three heuristics for processor allocation – Naive, Greedy, and Tree, into our structure-driven compiler. Figure 6 illustrates the operation of the three heuristics on the tree structured dataflow graph used in Section 1.2. As before, let the length of macro node i be L_i. As denoted by the size of the circles representing each macro node, macro nodes 1 and 3 (matrix products) take much longer to compute than the others (adds), and are the same length.

The Naive heuristic (Figure 6(a)) runs each macro node on all the available processors. Thus, macro nodes 1, 2, 3, 4, and 5 are run in sequence. In a program with multiple loop nests, Naive will allocate all the processors to each loop in turn. Although the execution times with this heuristic are clearly sub-optimal, this heuristic is used for its simplicity.

The Greedy heuristic (Figure 6(b)) is an as-soon-as-possible greedy (generalized) schedule. An macro node is run at the earliest time at which it is ready. All macro nodes that are ready at a certain time are started together and finished together. Computation proceeds as a wavefront picking up macro node sets that get ready in succession. For this expression, Greedy runs macro nodes 1 and 2 in parallel, distributing the processor resources among them such that they finish together. The resulting processor assignments are

$$p_1 = \frac{L_1^{1/\alpha}}{L_1^{1/\alpha} + L_2^{1/\alpha}} P \qquad \text{and } p_2 = \frac{L_2^{1/\alpha}}{L_1^{1/\alpha} + L_2^{1/\alpha}} P$$

(rounded to nearest integers). Subsequently, macro nodes 3, 4 and 5 are computed, each using all available processors ($p_3 = p_4 = p_5 = P$). Notice that since $L_1 >> L_2$, almost all processors are assigned to macro node 1 in the first step. The resulting schedule is little better than the Naive schedule. Indeed, if p_2 gets rounded to 0, all processing power is assigned to macro node 1 in the first step, and macro node 2 will be computed in a succeeding step. The Greedy schedule will then be identical to the Naive schedule.

The Tree heuristic (Figure 6(c)) does a much better job of partitioning the processor resources by recognizing that macro nodes 2, 3, and 4 form a subtree, which can be run in parallel with macro node 1, and splitting the available processors. The processing resource is split among the subtrees so that their finishing times are equalized. Finally, macro node 5 is run on all the available processors. The processor assignments are

$$p_1 = \frac{L_1^{1/\alpha}}{L_1^{1/\alpha} + (L_2 + L_3 + L_4)^{1/\alpha}} P$$

$$p_2 = p_3 = p_4 = \frac{(L_2 + L_3 + L_4)^{1/\alpha}}{L_1^{1/\alpha} + (L_2 + L_3 + L_4)^{1/\alpha}} P \qquad \text{and} \qquad p_5 = P$$

Since $L_1 \approx L_3 >> L_2, L_3, L_4$, both subtrees get roughly the same number of processors, and are computed in a "balanced" manner. Notice that the partitioning is greatly improved using the global information available about task sizes.

4 Optimal Matrix-Operator Compilation

Section 3 presented methods for determining the number of processors, p_i, assigned to each macro node i, and their sequencing. In our context, the macro nodes are matrix operators. This section presents our technique to optimally partition and schedule the dataflow graphs of each operator among the p_i processors. The analysis yields the speedup functions needed for processor allocation. Other methods (e.g., [1]) for general loop nests can also be used for partitioning.

Optimal matrix-operator routines are derived by exploiting the regular structure in the operator dataflow graph. Section 4.1 discusses how these matrix operator dataflow graphs can be represented as regular polyhedra. The basic idea is to represent the dataflow graph as a lattice, with each dataflow graph node corresponding to some lattice point. Locality in the dataflow graph is reflected in the geometric locality of the lattice points. Nodes corresponding to adjacent lattice points generally have common inputs, contribute to common outputs, or communicate values between themselves.

When the dataflow graph of each operator displays such regular structure, partitioning and scheduling can be based on geometric bin packing techniques, thereby resulting in better compilation times than general purpose approximate solutions to the NP-hard dataflow graph partitioning and scheduling problem. We illustrate in Section 4.3 the bin-packing method using matrix multiplication as an example.

4.1 Polyhedral Lattice Representation of Matrix-Operator Dataflow Graphs

The standard algorithm for multiplying an $N_1 \times N_2$ matrix A, by an $N_2 \times N_3$ matrix B, yielding an $N_1 \times N_3$ matrix $C = AB$,

$$c_{ik} = \sum_j a_{ij} b_{jk}$$

has $N_1 N_2 N_3$ multiplications, and $N_1 N_3 (N_2 - 1)$ additions. The corresponding dataflow graph can be represented by an $N_1 \times N_2 \times N_3$ lattice of multiply-add nodes, as shown in Figure 7(a). Each node (i, j, k) represents the computation

$$c_{ik} \leftarrow c_{ik} + a_{ij} b_{jk}$$

An element of A, namely a_{ij}, is broadcast to the N_3 multiply-add nodes having the same value of ij. These nodes are arranged in a line parallel to the k axis. Thus the computation on all nodes in this line exhibits *locality* with respect to the element a_{ij}. This broadcast of a_{ij} is represented by a solid line in Figure 7(b). Similarly, nodes having the same value of jk share b_{jk}. This broadcast of b_{jk} is also represented by the solid vertical line in Figure 7(b). Nodes having the same value of ik sum together to yield the same output element of C, namely c_{ik}. This accumulation is denoted by the dotted horizontal line in Figure 7(b). Every possible ordering of the computation, exploiting associativity and commutativity, can be represented by the geometric lattice.

Because of the existence of associativity and commutativity, the partial products can be summed in any order, and the accumulates into the output matrix C do not impose any precedence constraints, but must be atomic. Therefore, scheduling each chunk of nodes after partitioning is a non-issue.

Now, consider the cluster of nodes (for processor i) shown in Figure 7(c). The total number of input elements of matrix A accessed from global memory by this cluster, can be measured by the projected area of the cluster (PA_i) on the A face of the dataflow graph. Note that this presupposes the existence of fast local memory (cache) to reuse an already accessed datum. The same applied to matrix B. Thus PA_i and PB_i measure the total number of inputs accessed by this cluster. Similarly, the total number of output elements of matrix C to which the cluster contributes is measured by the projection on the C face of the dataflow graph (PC_i). Partial sums for each element in PC_i are formed inside this cluster, in local memory, and accumulated. Thus PC_i measures the number of synchronized accumulates (*fetch-and-adds*), due to this cluster. Hence the communication of this cluster with the outside world can be minimized (or the locality maximized) by minimizing the projected surface area on the three faces, which can be handled by geometric techniques. A cluster that exhibits geometric locality (in terms of minimal projected surface area) exhibits dataflow graph locality.

To summarize, the dataflow graph of a matrix multiply can be represented by a 3-dimensional polyhedral lattice, each lattice point representing a computation in the dataflow graph. Adjacent lattice points share some common broadcast inputs or are accumulated to common outputs. Dataflow-graph locality is equivalent to geometric locality. Similar representations can be derived for matrix sums, inverses, FFT's, LU decompositions, etc, from the general shapes of their computational lattices.

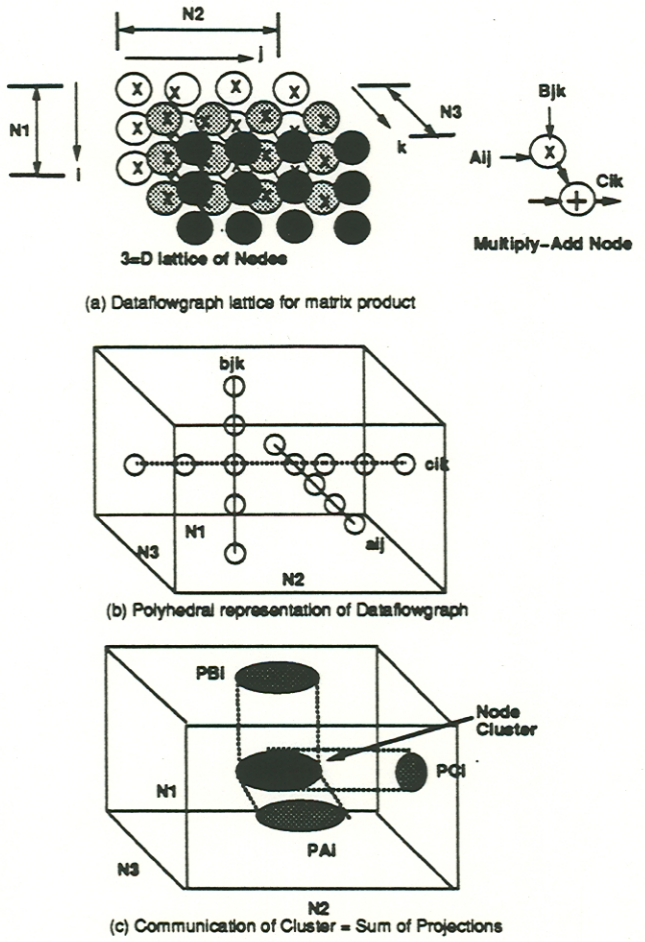

Figure 7: Dataflow graphs for matrix product

4.2 Performance Metric

The polyhedral lattice representation facilitates partitioning to minimize communication using geometric methods. These methods require a cost metric to evaluate potential choices. Although the cost of any partition and associated schedule is ideally the execution time, its accurate estimation at compile time in a real system is a difficult problem. We therefore approximate the execution time T_e of a partition i by the sum of the maximum number of operations of each kind, namely, adds, multiplies, remote fetches, and synchronized accumulates, weighted by their associated costs (from Table 1), in that partition. (Recall, local access costs are included in the add and multiply times.) In other words, if N_a is the maximum number of additions performed by any processor, N_m the maximum number of multiplications performed by any processor, N_u the maximum number of unsynchronized remote data fetches (in words), and N_{fa} the maximum number of synchronized accumulates (in words), we have,

$$T_e = T_a N_a + T_m N_m + T_u N_u + T_{fa} N_{fa} \tag{2}$$

Our cost metric is accurate if the processors expend negligible time waiting for work. This assumption is valid for matrix arithmetic (except at very fine task granularity), because of the extensive parallelism available. For example, for matrix sums and products, all nodes are independent. The chunks corresponding to different processors can all be scheduled in parallel, without embedded synchronization (except for that implicit in the accumulates).

4.3 Optimal Matrix Products

The geometric dataflow graph representation simplifies the partitioning, by enabling us to derive a lower bound on the execution time, and bin-packing heuristics to closely approach that bound. In this section we derive the lower bound. The next section presents the bin-packing heuristics.

An optimal partition (of the i^{th} operator) on p_i processors divides the operator dataflow graph into p_i roughly equal sized chunks, while choosing the shape of the chunks in a compact manner to minimize communication. For convenience, we will drop the subscript i, keeping in mind that p is the number of processors allocated to macro node (operator) i. Keeping the chunks equal in size balances the workload, minimising N_a and N_m. Compact chunks minimize the number of data transfers, N_u and N_{fa}, and hence the communication. The ideal cluster shape can be shown to be a cubical box, whose aspect ratio depends on the architectural parameters T_u (unsynchronized memory access time) and T_{fa} (synchronized accumulates).

Assuming that load balancing is perfect, and all clusters are ideal in shape and can be scheduled concurrently, a lower bound on the total execution time, T_e, for the matrix product is

$$T_e \geq (T_a + T_m)\frac{N_1N_2N_3}{p} + 3(T_u^2T_{fa})^{\frac{1}{3}}(\frac{N_1N_2N_3}{p})^{\frac{2}{3}} \qquad (3)$$

The first term represents the arithmetic operation cost for each cluster, assuming the $N_1N_2N_3$ adds and multiplies are evenly distributed over the P processors. That is, the volume of the cluster is fixed at $\frac{N_1N_2N_3}{p}$ for an even load. The second term results from choosing the aspect ratio of the cluster so as to minimize the data access time, and can be derived as follows.

Recalling, each partition issues N_u unsynchronized remote memory accesses to each of the input matrices A and B, and issues N_{fa} *fetch-and-adds* for the accumulates into C. Hence the total time for the data accesses is

$$T_uN_u + T_{fa}N_{fa}$$

The data access time thus depends on relative magnitude of N_{fa} and N_u, or the aspect ratio of the cluster.

Let the ideal cluster dimensions be I, J, and K ($I = K$ because accesses of matrices A and B incur the same cost). Then $N_u = 2IJ = 2JK$ and $N_{fa} = IK$. Thus, the problem of obtaining N_u and N_{fa} to minimize the time for data accesses is reduced to the problem of determining I, J, and K, such that

$$2IJT_u + IKT_{fa}$$

is minimized, subject to the constraint

$$IJK = \frac{N_1N_2N_3}{p}$$

The above constrained minimization problem can be solved using Lagrange multipliers to yield optimal values of I, J, and K, from which the minimal communication cost can be derived, yielding the result in Equation 3.

When $N_1 = N_2 = N_3 = N$ and $T_{fa} = 2T_u$, we get

$$T_e \geq (T_a + T_m)\frac{N^3}{p} + \frac{3.8T_uN^2}{p^{\frac{2}{3}}}$$

The lower bound on the total time has a linearly decreasing (strictly inversely linear) and a less than linearly decreasing component, with respect to the number of processors p. The maximum attainable speedup $S(p)$, which is the ratio of the execution time on 1 processor to the execution time on p processors, then becomes

$$S(p) = \frac{(T_a + T_m)N^3 + 3.8T_uN^2}{(T_a + T_m)\frac{N^3}{p} + \frac{3.8T_uN^2}{p^{\frac{2}{3}}}} \tag{4}$$

As we increase the number of processors, the maximum attainable incremental speedup keeps decreasing. The speedup function provides a simple characterization of the optimal matrix operator routines for use in the processor allocation phase (Section 3).

4.4 Bin Packing Algorithms

In general the lower bound determined in Equation 3 cannot be met exactly, as p ideally shaped clusters cannot be packed into the cuboidal dataflow graph. In such situations, the clusters must be distorted so they exactly fill the dataflow graph. Fortunately, because the projection sum (surface area) does not vary much with cluster shape for clusters near the optimum, good partitions can be achieved even if some or all the clusters are slightly distorted. This observation is the basis for close to optimal bin packing algorithms used in the compiler.

Our techniques are best illustrated using the the 2-D problem of partitioning an $N_1 \times N_2$ rectangle into P equal sized chunks to minimize the sum of projections. For this 2-D problem in the continuous domain, simple algorithms have been shown to be [10, 11, 12, 13]. Mount, Kong, and Roscoe [13] have applied these results as approximations to discrete 2-D lattices as well. We have extended these algorithms to three dimensions [14]. We shall now describe the continuous algorithms, as they are much simpler. The discrete algorithms usually start in the same manner as the continuous algorithms, but are followed by a discretization step.

For simplicity, assume that $T_u = T_{fa}$, so that all projections are weighted equally and the exact projection sum is being minimized. Since all p chunks are equal in size, the area of each chunk is fixed. To minimize the projection sum, the projection sum of each chunk should ideally be minimized, keeping its area fixed. This implies that each chunk should ideally be a square. In general, however, p equal area squares cannot be exactly fitted into an $N_1 \times N_2$ rectangle. The 2-D partitioning algorithms fix this by distorting the ideal square chunks so as to fit inside the $N_1 \times N_2$ rectangle.

These algorithms start by arranging p squares to roughly fill the $N_1 \times N_2$ rectangle. The resulting polygon will approximate a rectangle, possibly with one side incomplete, and heavily

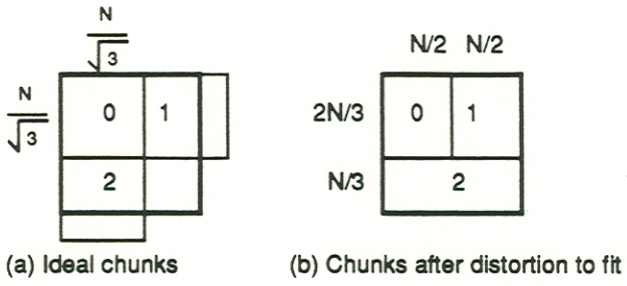

Figure 8: Optimal square partitioning.

overlap the original $N_1 \times N_2$ rectangle. Next all P chunks are distorted roughly equally, in such a manner that the $N_1 \times N_2$ rectangle is exactly filled. In practice, the initial arrangement of the squares is critical to success, while the distortion technique used does not matter as much, since the projected surface area does not increase greatly with distortion.

For example, consider partitioning an $N_1 \times N_1$ square into three pieces. Figure 8(a) shows the ideal three square pieces (assume $T_u = T_{fa}$), arranged such that they approximately fill the square. Figure 8(b) shows the optimal partition obtained after distorting the pieces to exactly fill the square. Since small changes in aspect ratio of any piece does not change its projection sum very much, the projection sum in Figure 8(b) is close to that in Figure 8(a). If $T_u \neq T_{fa}$, then the weighted projection sum is minimized, the aspect ratio of the ideal chunks is not $1:1$, and the ideal chunks are rectangles. These ideal rectangular chunks are distorted in the same way, to fit inside the $N_1 \times N_2$ rectangle. Small distortions do not cause large increases in communication.

The bin packing techniques used in SDC are generalizations of these techniques to 3-D $N_1 \times N_2 \times N_3$ lattices. It should be noted that 3-D bin packing is extremely tedious if the number of chunks (processors), is not well matched to the problem size. For example, chunking up an $32 \times 32 \times 32$ matrix product dataflow graph into 25 chunks results in a variety of chunks with different aspect ratios and locations, which are very difficult to compute by hand. The automated procedure in SDC is essential to perform this task.

4.5 Other Operators

Various other matrix expression operators have been implemented, following the same basic idea of minimising communication by choosing good block shapes. These operators include matrix sums, dot products, outer products, matrix scaling, transposes, etc.

5 Implementation Details

This section furnishes details of the experimental environment used to evaluate the performance of the matrix expression compiler, SDC, as well as some implementation details.

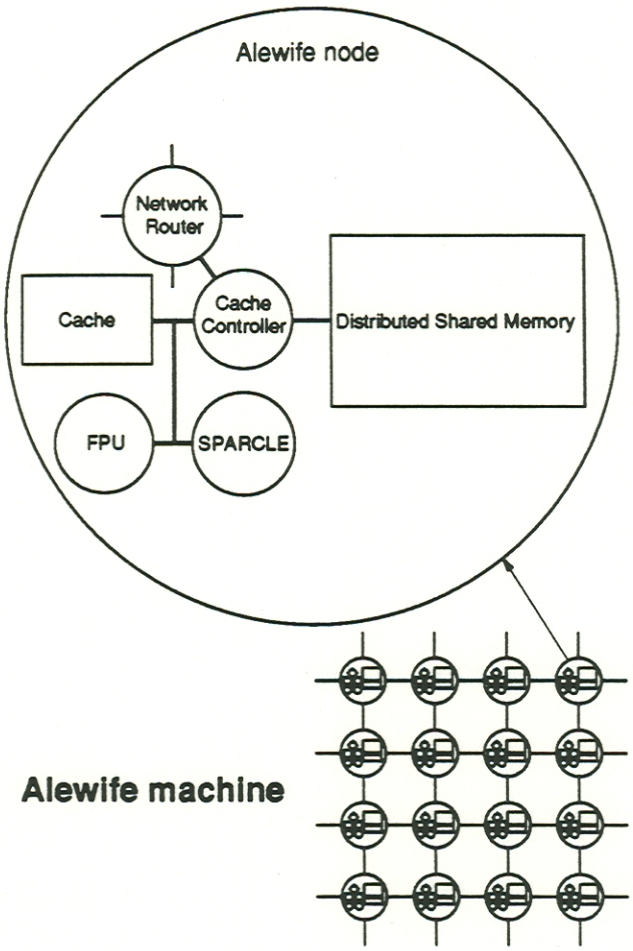

Figure 9: The Alewife machine.

5.1 Experimental Environment

The input to SDC is a matrix expression in a LISP-like prefix language, with data-independent control. The compiler produces a parallel program in Mul-T [15], which is compiled and run on the Alewife machine simulator, ASIM. Mul-T is a parallel lisp language.

The Alewife Machine The Alewife machine [6] is a mesh-connected, distributed shared-memory multiprocessor, with coherent caches, as shown in Figure 9. The processor, called Sparcle has a modified SPARC architecture. Global shared-memory is distributed among the processing nodes, access to which is provided by the mesh interconnection network. Processors have associated caches for fast access to frequently used data. Because caches can store shared data, a cache coherence scheme maintains memory consistency. A detailed, cycle-by-cycle simulator, ASIM, and associated program analysis tools, are currently available for Alewife.

The abstractions made in Section 2.2 model Alewife fairly accurately. First, the Sparcle

processor uses a load-store architecture, so arithmetic operations are accurately characterized by the execution times (T_a, T_m), ignoring pipeline latency. The local cache in each Alewife node allows single-cycle word accesses. As mentioned earlier, cache access costs are included in the basic arithmetic operation costs.

Accesses to globally shared memory, however, are not accurately represented, since Alewife's distributed-memory architecture places some memory modules closer to each node than others – the access time, T_u, to the global memory module on the same node as the requesting processor is satisfied in 10 cycles, while an access to a memory module situated in a remote node could take on the order of 50 cycles (in a 64-processor system). Access times also vary due to network loading conditions, hotspots, etc. However, because each of these access times is over an order of magnitude greater than the cache access time, the compiler can reasonably model Alewife as a two-level memory hierarchy, with a fast single-cycle first-level store, and a much slower second-level remote memory.

A coherence scheme in Alewife ensures data consistency, but we do not explicitly model its effects. The presence of a cache coherence scheme manifests itself in our approximation of accumulation cost $T_{fa} = 2T_u$. Because it requires a write to a shared data location potentially present in another cache, an accumulation incurs roughly twice the cost of a memory read due to the extra invalidate to purge the other cached copy. For the results in this paper we configure ASIM to use the full-map coherence protocol, which keeps track of all cached copies of a shared datum.

Alewife's Software Environment The SDC algorithms determine an optimal allocation of work (thread) for each processor over time, for computing the matrix expression. Code generation for the work allocation requires the ability to spawn threads for each processor, synchronize them, and communicate data (input and output matrices and temporaries). The Alewife software environment allows thread creation using the `future` call. A thread t can be assigned to a specific processor p using the (`future-on` p t) call. Threads can be synchronized using various constructs, including distributed semaphores, provided by the Alewife parallel software library. Input and output matrices and temporaries are automatically shared among multiple threads through shared memory.

5.2 Compiler Implementation

The compiler performs a two level partitioning and scheduling. In the first step (processor-allocation), the sequencing and parallelism of all the matrix operators is determined using their speedup functions, as described in Section 3.4. While the dataflow graph analysis of Section 4 can be used to estimate these speedup functions, SDC approximates all speedup functions to be of the form P^α to simplify the scheduling (Section 3.5). Although experimentally determined α's have been used for the measurements, we show that the results do not differ materially with small variations in the value of α.

Next, given the parallelism, each operator dataflow graph is partitioned using bin packing algorithms, as described in Section 4. This partitioning phase depends on architectural parameters – the computation times T_a and T_m, and data communication times T_u and T_{fa}.

These partitions are grouped into P threads, one for each processor. Finally, code consisting of a sequence of calls to the routines handling the operator partitions (the partition library) is generated. Sets of processors cooperating on a macro node are synchronized by embedding distributed semaphores at appropriate places in each thread.

A distributed-memory multiprocessor, such as Alewife, requires that shared data structures like matrices and vectors be distributed across multiple processors, and necessitates a mechanism for keeping track of the constituent chunks. Since the data sizes can be arbitrary, and ill-matched to the sizes of the multiprocessor, the sizes of the chunks are usually irregular. We provide access through pointers to the chunks. The pointers are duplicated in some of the processor nodes to avoid hot spots. Because these features complicate bookkeeping, making use of the automatic procedures in SDC is highly desirable.

5.3 Design of the Partition Library

The optimal matrix operator partitioning algorithms (Section 4.3) result in box-like partitions, whose aspect ratios are a function of the communication costs. We have developed a library of routines for various matrix operators that accept as arguments the input and output matrices and the index limits over which the computation is desired. For efficiency, separate routines are required for distinct shapes. We discuss the design of this library and associated data structures below.

Consider the routine for cuboid (box like) partitions of a matrix multiply, *mul_block_loop*, as depicted using pseudo code in Figure 10. When a matrix product is computed on P processors, the thread assigned to each processor calls *mul_block_loop* with arguments specifying the three matrices, *mat_a mat_b*, and *mat_c*, and the parameters, *starti, endi, startj, endj, startk*, and *endk* specifying its partition size. The routine computes partial sums, and then accumulates the partial sums to the output matrix. The partition computed by a processor is specified by

$$[i, j, k] \geq [starti, startj, startk] \qquad \text{and} \qquad [i, j, k] < [endi, endj, endk]$$

The accumulate necessitates synchronization among all processors computing an element of the output matrix. In Alewife, the availability of data structures with built-in synchronization, such as L-structures [16], greatly facilitates this task. Because an L-structure allows efficient exclusive access to each of its elements, an atomic *fetch-and-add* operation can be efficiently performed on each of its elements, without locking the whole structure. SDC, therefore, represents matrices as L-structures.

```
mul_block_loop (mat_a mat_b mat_c starti endi startj endj startk endk)
 do i from starti to endi
   {
   do k from startk to endk
     {
        sum=0;
        do j from startj to endj
           sum = sum + a[i,j] b[j,k];
        fetch-and-add (c[i,k] sum);
     }
   }
```

Figure 10: Routine to compute block of matrix product.

6 A Detailed Example

Figure 11(a)) illustrates the operation of SDC using the expression $g1$ first introduced in Section 1.2, using the Tree heuristic. All matrices in the expression (see below) are of size 32×32

$$
\begin{array}{l}
(+\ (\times\ A_0\ A_1) \\
\quad (+\ (\times\ (+\ A_2\ A_3) \\
\qquad\quad A_4) \\
\qquad A_5))
\end{array}
$$

First, the Tree heuristic is used to determine the sequencing and parallelism of each macro operator, using $\alpha = 0.7$. The selection of α is based on speedup measurements presented in Section 7. Figure 11(b) shows a Gantt chart for the resultant schedule. The 64 processors available are allocated to the two branches of the tree such that they are started and finished together. The values of $p_1 \ldots p_5$ are computed as indicated in Section 3.6.

Since $L_1 = L_3 >> L_2, L_4$, the processor allocations to the branches are almost equal, with 31 processors handling the product $A_0 \times A_1$ and 33 processors computing each operator in the other branch. Finally, all 64 processors cooperate to compute the final addition (Addition 5).

Next, each of the operator dataflow graphs is optimally partitioned for the parallelism determined above. Multiply 1 is partitioned into 31 blocks, while Additions 2, Multiply 3, and Addition 4 are partitioned into 33 blocks each. The final addition 5 is partitioned into 64 blocks. Finally, the blocks are grouped into 64 distinct sets, each set yielding a thread of computation for a processor. The thread is formed by repeated calls to the routines computing the dataflow graph blocks (Section 5.3). Synchronization in the form of distributed semaphores is embedded in the threads automatically. The details of two threads, viz. those produced for processors 0 and 60, are described below.

The thread for processor 0 works on Multiply 1 and Addition 5. It computes the following dataflow graph blocks, in order ((i,j,k) refer to dataflow graph indices - Section 4).

Multiply 1: $[i,j,k] \geq [0,0,0]$ and $[i,j,k] < [8,17,8]$
Addition 5: $[i,j] \geq [0,0]$ and $[i,j] < [4,4]$

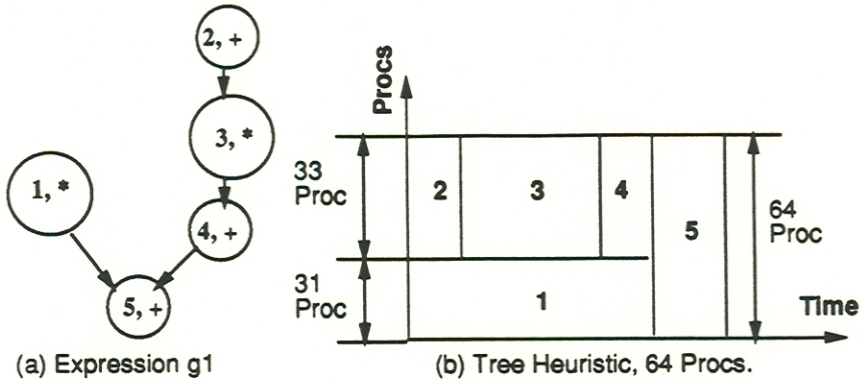

Figure 11: Tree schedule for $g1$.

Since $T_{fa} = 2T_u$ the matrix product blocks are close to cubical to minimize communication, with the $N_2(j)$ dimension slightly larger than the other. A synchronization point (semaphore) is inserted before Addition 5, to ensure that both branches of the tree are finished before Addition 5 is begun.

Similarly, the thread for processor 60 works on the larger branch of the tree, and computes the following blocks, in order.

Addition 2:	$[i,j] \geq [27,22]$	and	$[i,j] < [30,32]$
Multiply 3:	$[i,j,k] \geq [24,16,0]$	and	$[i,j,k] < [32,32,8]$
Addition 4:	$[i,j] \geq [27,22]$	and	$[i,j] < [30,32]$
Addition 5:	$[i,j] \geq [28,16]$	and	$[i,j] < [32,20]$

The code produced by SDC is compiled and run on Alewife. Performance results are presented in Section 7.

7 Experimental results

This section presents experimental results obtained with our compiler on the Alewife machine simulator. A large number of matrix expressions have been compiled and simulated. The simulator is configured for a three-dimensional mesh interconnect, with sizes $1 \times 1 \times 1$ (uniprocessor), $2 \times 2 \times 2$, and $4 \times 4 \times 4$. If the number of processors used to compute the matrix expression is P, then processors P_i such that $0 \leq i < P$ are used and the others are idle. Unless otherwise specified, the compiler assumes that the time for an interprocessor accumulate, T_{fa}, is twice the memory access time, T_u.

We present results for the the matrix product first (other operators are analogous), and show how speedup functions are derived. Then, using these speedups, we present results for complete expressions. We emphasize that all the speedups shown are close to the "true" speedups for the expression. That is, the uniprocessor program is optimized for a single processor, that for 8 processors is optimized for 8 processors, and so on.

$T_{fa} = T_u$	$T_{fa} = 2T_u$	$T_{fa} = 4T_u$	Long Blocks	One Row Per Proc
289823	292376	292376	310358	895758

Table 2: Time (cycles) for 64×64 matrix product, using 64 processors.

7.1 Matrix Product

Code for a matrix product is generated by the compiler for each number of processors, following the bin-packing techniques of Section 4. We report on four sets of experiments with matrix products:

1. Since our techniques depend on the access times T_u and T_{fa}, this section first investigates the sensitivity of the partitioning to these parameters.
2. We then evaluate the partitioning scheme when the number of processors is mismatched to the dimensions of the operator dataflow graph.
3. Because our (compile time) scheduling methods rely on estimated speedup functions, we assess the accuracy of our matrix product speedup function.
4. We show that the matrix product speedup curve can be reasonably approximated by a speedup function of the form $s(P) = P^{\alpha}$, which justifies the use of simple, close-to-optimal processor allocation algorithms.

Table 2 shows the time taken to compute a 64×64 matrix product on 64 processors, with varying choices for the architectural parameters, T_u and T_{fa}. Each choice of the parameters results in a different shape of the dataflow-graph blocks computed by each processor. Also shown is the execution time when T_{fa} is so large that the compiler places each output matrix element to be computed on a single processor (Long Blocks). In this case the output matrix is split into P square blocks, each computed on one processor. The resultant communication is $\Theta(\sqrt{P})$, instead of $\Theta(P^{1/3})$ (Equation 3). Also, shown is the execution time when the matrix-product dataflow graph is sliced into P slices along the i dimension only (that is, by rows). Then, we have an entire row of the output matrix computed on a single processor. The communication is then $\Theta(P)$ versus $\Theta(P^{1/3})$ for the optimal blocking, which is much worse.

The results show that the execution time is not very sensitive to the architectural parameters, as long as they are reasonably accurate. This is because the communication does not increase very much if the shapes of the blocks are not very far from optimal. Small changes in the architectural parameters have a correspondingly small effect on the block shape chosen by the bin packing algorithms. Notice however, the Long Blocks partition is definitely worse than the first three. The performance with one row computed per processor, is a factor of 3 worse than the first three.

Next, let us assess the performance of the partitioner when the number of processors is mismatched to the matrix sizes. Such mismatched numbers are often encountered after the processor allocation phase (Section 3). Figure 12 shows the speedup curves for the product of two matrices with varying sizes, on 1, 8, 16, 25, 32, 38, and 64 processors. Note that two numbers (25 and 38), are mismatched to the matrix sizes.

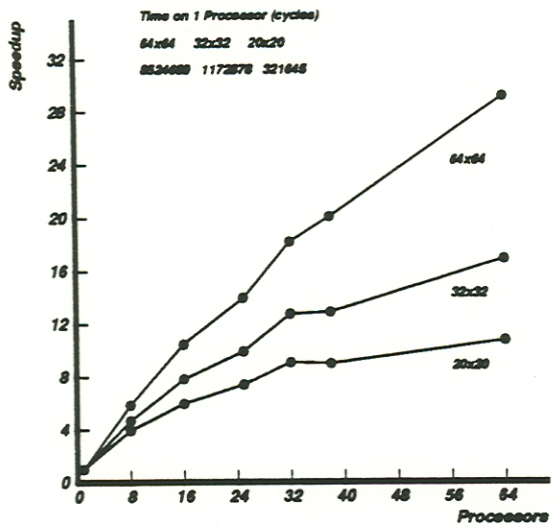

Figure 12: Speedup curves for matrix product.

We observe that our algorithms work well for ill matched processor allocations. For example, the speedup for 25 processors is about midway between the speedups for 16 and 32 processors. In general, however, the mismatch causes a slight drop in speedup due to imperfect bin packing. The mismatch is particularly noticeable for 38 processors. However, this effect decreases as the matrix size increases, as is shown by the curve for 64 processors in the figure.

To achieve the speedup for 25 processors for the $32 \times 32 \times 32$ product, SDC's bin packing algorithms produce partitions with a variety of sizes: 12 partitions of size $8 \times 16 \times 11$, 6 partitions of size $8 \times 16 \times 10$, 4 partitions of size $8 \times 18 \times 8$, 2 partitions of size $8 \times 14 \times 11$, and a single partition of size $8 \times 14 \times 10$. Clearly, it would be extremely difficult for a programmer to determine all these partitions by hand.

We now attempt to assess the accuracy of our estimated speedup function. Consider the formula for the execution time in Section 4.3 (Equation 3), repeated here for convenience.

$$T_e = (T_a + T_m)\frac{N_1N_2N_3}{P} + 3(T_u^2T_{fa})^{\frac{1}{3}}(\frac{N_1N_2N_3}{P})^{\frac{2}{3}}$$

We shall assess its accuracy by examining a slightly different representation of the formula. Notice that a plot of the product of the execution time and the number of processors ($P \times T_e$) versus $P^{1/3}$ should ideally be a straight line, as is evident by rewriting the above equation as

$$\begin{aligned} P \times T_e &= (T_a + T_m)N_1N_2N_3 + 3(T_u^2T_{fa})^{\frac{1}{3}}(N_1N_2N_3)^{\frac{2}{3}}P^{\frac{1}{3}} \qquad (5) \\ &= k_1(N_1N_2N_3) + k_2(N_1N_2N_3)^{\frac{2}{3}}P^{\frac{1}{3}} \end{aligned}$$

for $k_1 = (T_a + T_m)$ and $k_2 = 3(T_u^2T_{fa})^{\frac{1}{3}}$. Figure 13 shows the $P \times T_e$ versus $P^{1/3}$ plots for matrix products of various sizes. The plots are well approximated by straight lines, with the

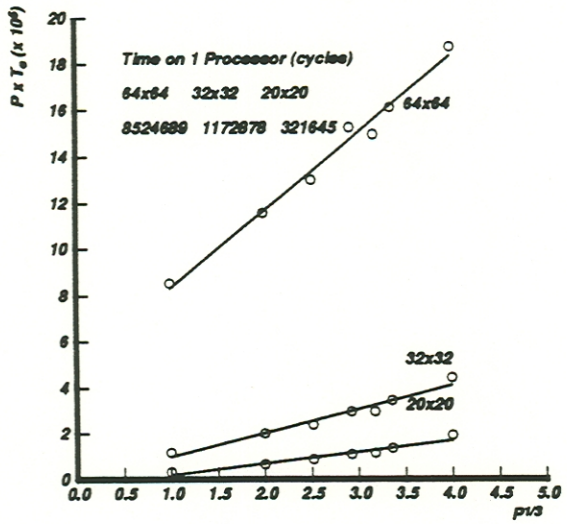

Figure 13: $P \times T$ versus $P^{1/3}$ curves for matrix product.

fit improving as the matrix size increases from 20×20 to 64×64.

The processor allocation (or (generalized scheduling) techniques of Section 3 require the operator speedup functions to be of the form P^α, for some α. Below, we investigate whether the speedup function for matrix multiply can be reasonably characterized by a function of the form P^α for various matrix sizes. The speedup function for matrix products can be derived from Equation 6 as,

$$S(P) = \frac{k_1 N_1 N_2 N_3 + k_2 (N_1 N_2 N_3)^{\frac{2}{3}}}{k_1 \frac{N_1 N_2 N_3}{P} + k_2 \left(\frac{N_1 N_2 N_3}{P}\right)^{\frac{2}{3}}} \tag{6}$$

While is is not immediately apparent that this speedup function is of the form P^α, it can be approximated as such, as is evident from Figure 14. This figure shows a log-log plot of the speedup curves for various matrix sizes. Functions of the form P^α will appear as straight lines on a log-log plot, whose slope is the desired parameter α. Since all the curves are roughly straight lines, they are well approximated by P^α.

The slopes (α's), however, depend on the size of the matrices, and range from from 0.6 (for 20×20 matrices) to 0.8 (for 64×64 matrices). Fortunately, as we demonstrate below, the partitioning is not very sensitive to the exact value of α used, as long as the task sizes are not widely different. An average value of $\alpha = 0.7$ can be used for the matrix sizes above. This is in rough agreement with Equation 6, which implies that α should lie between 2/3 and 1.

The value of α can also be estimated from a knowledge of the multiprocessor constants T_a, T_m, and T_{fa}. However, the presence of stochastic synchronization behavior in T_{fa} precludes its accurate apriori estimation. Alternatively, measurements of execution times for two or three values of P allows us to estimate k_1 and k_2 in Equation 6, thus enabling the matrix product speedup function to be determined. Although it is not necessary, linear regression can be used

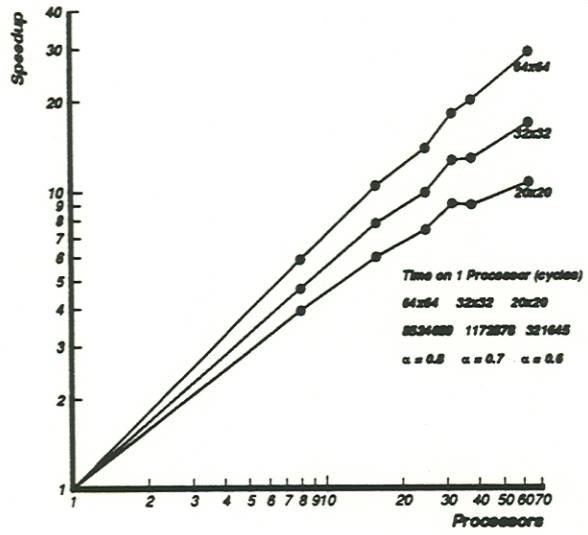

Figure 14: Speedup curves for matrix product (drawn using log-log scales).

for higher accuracy.

7.2 Almost Balanced Tree – $g1$

We now evaluate the performance of the compiler on several matrix expressions for various matrix sizes. We first present results for the almost-balanced tree expression $g1$ first introduced in Section 1.2,

$$
\begin{array}{l}
(+\ (\times\ A_0\ A_1) \\
\quad (+\ (\times\ (+\ A_2\ A_3) \\
\qquad\quad A_4) \\
\qquad A_5))
\end{array}
$$

with matrix sizes 32×32, and use $\alpha = 0.7$ in the scheduling heuristics.

The speedup curves plotted in Figure 15 show the performance of the Naive, Greedy, and Tree heuristics as the number of processors is increased from 1 to 64. We also plot the expected speedup for the Tree heuristic assuming $\alpha = 0.7$, and assuming no costs are associated with the synchronization needed for enforcing the precedence constraints. The expected speedup (for P large enough to ignore processor discretization effects) is computed as follows

$$S(P) = \frac{T(1)}{T(P)}$$

where $T(1)$ is the expected uniprocessor execution time, and $T(P)$ is the expected time for the Tree heuristic. From Section 3.5 it follows that

$$T(P) = \frac{\text{Equivalent Tree Length}}{P^{0.7}}$$

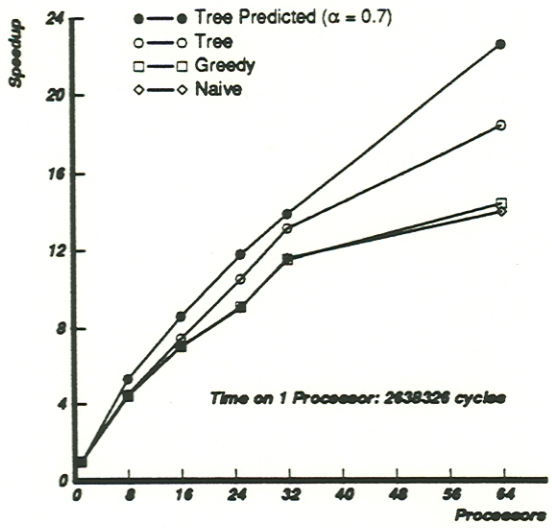

Figure 15: Speedup curves for expression $g1$, with 32×32 matrices.

α	0.5	0.7	1.0
Tree	19.1	18.6	18.6
Greedy	14.5	14.5	7.6

Table 3: Speedup for 64 processors, using various values of α.

where the equivalent tree length is computed using the series parallel reductions of Section 3.5, and is always less than the uniprocessor execution time.

The Tree heuristic is clearly the best, and gets progressively better relative to Naive and Greedy as more processors are used. The Naive and Greedy heuristics are identical in performance for $g1$; their speedup flattens out at about 14 after 32 processors. Thus, compilers for programs with multiple loop nests which assign all processors to each loop nest are potentially far from optimal. The performance of the Tree heuristic is much better; the absolute speedup for the Tree heuristic at 64 processors is about 18, a gain of 30 percent over the Greedy and Naive heuristics. This is close to 80 % of the expected speedup of 23. This gap between the expected and measured speedup is largely due to synchronization costs associated with enforcing the precedence constraints.

How robust is the partitioning strategy to the value of α used? Answering this question is important because, in practice, it is hard to estimate α accurately. Table 3 shows the speedups obtained for $g1$ as α is changed from 0.5 to 1.0. It can be seen that the Tree heuristic is quite robust, with the speedup changing by less than 10 percent. Greedy, however, shows a larger sensitivity to α. This can be explained as follows.

The relative insensitivity of the Tree heuristic to α is because the two branches of the tree are almost equal in size, and are assigned about the same number of processors, irrespective of α.

Greedy, however, tries to run a large multiply, $(\times\ A_0\ A_1)$, in parallel with a small addition, $(+\ A_2\ A_3)$, in the first step. For small α, because the speedup curves are highly convex, the smaller task (addition) gets a tiny fractional processor assignment. After discretization, all the processors get assigned to the larger task, and the smaller task is computed in a succeeding step. Hence, when α is small, the two operations are computed one after another, in sequence. As α increases, there is a point at which the small addition gets a non-zero processor assignment in the first step. The small assignment for the addition causes it to finish late. Consequently, the sensitivity of the processor assignment to α is especially significant when the sizes of the tasks being run in parallel are widely different. Greedy can be made more robust by not running very small and very large operators concurrently. The robustness gained more than compensates for the small loss in efficiency.

7.3 Large Almost Balanced Tree – $g2$

The next example demonstrates that the relative gain in performance from the Tree heuristic increases with an increase in the parallelism in the macro dataflow graph. Expression $g2$ shown below is essentially two copies of $g1$ executed in parallel.

$$
\begin{array}{l}
(+\ (\times\ A_0\ A_0) \\
\quad (+\ (\times\ (+\ A_1\ A_1) \\
\qquad\quad A_1) \\
\qquad (+\ (\times\ (+\ (+\ A_2\ A_2) \\
\qquad\qquad\quad A_2) \\
\qquad\qquad A_2) \\
\qquad\quad (\times\ (+\ (+\ (+\ A_3\ A_3) \\
\qquad\qquad\qquad A_3) \\
\qquad\qquad\quad A_3) \\
\qquad\qquad A_3))))
\end{array}
$$

It has 4 matrix products which can be computed in parallel, as opposed to only 2 products for $g1$. This gives the Tree heuristic more opportunity to run tasks in parallel, with resultant higher gains.

The speedups are plotted in Figure 16. The Tree heuristic performs much better than Naive or Greedy, with gains increasing as we increase the number of processors. The speedup of 24.5 with 64 processors is 85 percent of the expected speedup of 28. This is a 60 % gain over the performance of 15.8 attained by Naive or Greedy.

7.4 Large Unbalanced Product Tree – $g3$

The expression $g3$,

$$
\begin{array}{l}
(\times\ (\times\ (\times\ A_1\ A_2) \\
\qquad (\times\ (\times\ A_3\ A_4)\ A_5)) \\
\quad (\times\ (\times\ A_6\ A_7)
\end{array}
$$

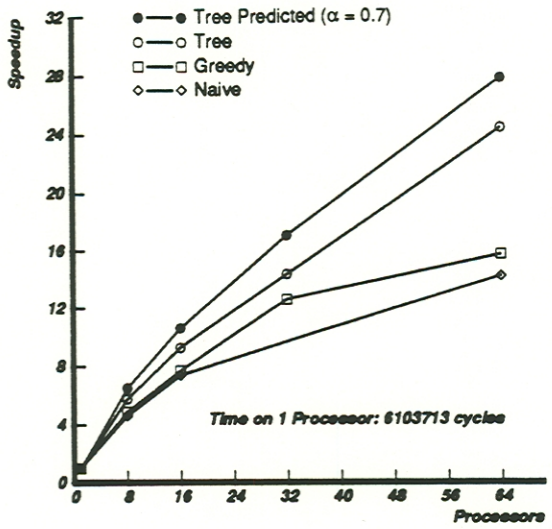

Figure 16: Speedup curves for expression $g2$, with 32×32 matrices.

$$(\times\ (\times\ A_8\ A_9)\ (\times\ (\times\ A_{10}\ A_{11})\ (\times\ (\times\ A_{12}\ A_{13})\ A_{14}))))$$

is an unbalanced tree, with one branch twice as large as the other. All matrices are of size 32×32. Since all the basic operators in this expression are matrix products, both Tree and Greedy have the opportunity to run large operators in parallel, resulting in significant gains in efficiency. This is unlike $g1$ or $g2$, where the presence of a tiny matrix sum in front of a matrix product prevented the Greedy heuristic from running the products in parallel.

Figure 17 shows the speedup curves for the Naive, the Greedy, and the Tree heuristics. The expected speedup for the Tree heuristic, for $\alpha = 0.7$, has also been plotted. The Tree heuristic performs slightly better than the Greedy. The Greedy and Tree heuristics are also substantially better than the Naive, by a factor of 1.5 for 64 processors. The curves are around 85 percent of the expected speedup for Tree.

This example demonstrates that the Greedy heuristic may approach the performance of Tree in specific cases. However, Greedy is not as robust as Tree, and does as poorly as Naive in many cases.

7.5 Non-Tree Expressions - Matrix Polynomials $g4$, $g5$

The results presented so far have demonstrated the superiority of the Tree heuristic for tree structured macro dataflow graphs. When the macro dataflow graphs are not trees, the simple Greedy heuristic is a reasonable approach, and is far superior to Naive in many cases.

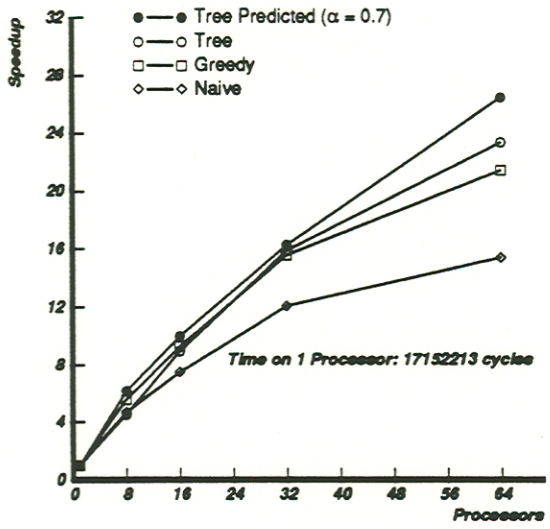

Figure 17: Speedup curves for $g3$, with 32×32 matrices.

We consider two matrix polynomials: $g4$ with degree 8, and $g5$ with degree 16.

$$g4: \sum_{i=0}^{8} c_i A^i$$

$$g5: \sum_{i=0}^{16} c_i A^i$$

These matrix polynomials are prototypical non-trees, since the powers of the matrices are generated in a recursive-doubling fashion. $g5$ has about twice the parallelism of $g4$ at the macro-node level.

Figures 18 and 19 show the speedup curves comparing Naive and Greedy for $g4$ and $g5$. It is clear from the figures that Greedy performs much better than Naive, with the gains increasing as we increase the number of processors. Furthermore, comparing the speedup for $g4$ and $g5$, it is evident that the gains increase as the parallelism in the macro dataflow graph increases. For $g4$, Greedy is a factor of 2 better than Naive, for 64 processors. This gain increases to between 3 and 4 for $g5$.

The Greedy heuristic can hence be employed as a general (but sometimes suboptimal) processor allocation strategy for sets of interdependent nested loops, when the precedences between loop nests are not tree structured.

7.6 Compilation Time

SDC's compilation time is $\Theta(M)$, where M is the number of macro nodes. By comparision, a compiler using general partitioning and scheduling techniques as in [17] takes time $\Theta(N)$

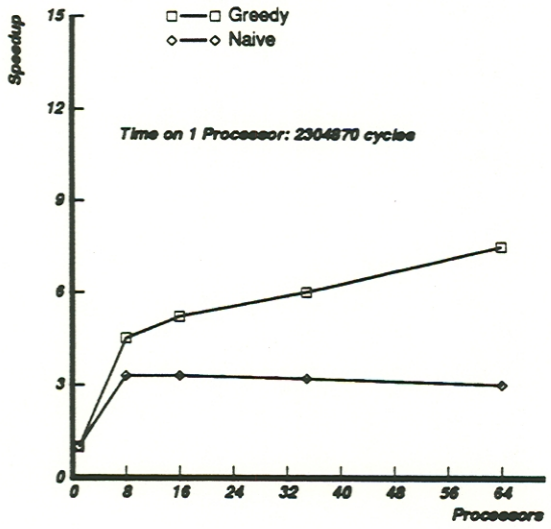

Figure 18: Speedup Curves for expression g4.

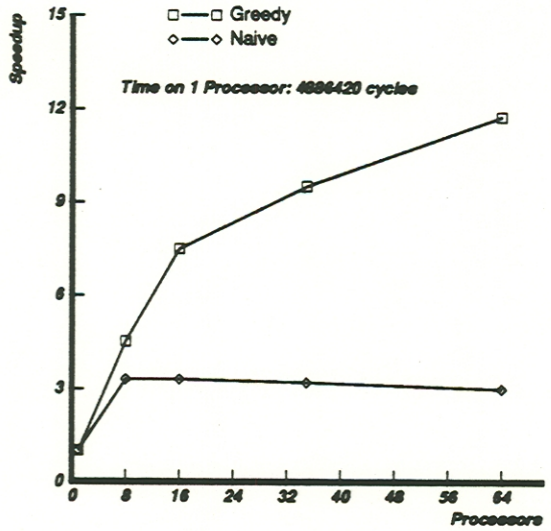

Figure 19: Speedup Curves for expression g5.

to $\Theta(N^3)$, where N, the number of expanded (simple) nodes, is frequently several orders of magnitude greater than M. Clearly the hierarchical scheme is much faster, when the structure of the computations can be exploited. For our example expression g_3 consisting of 13 operators, SDC takes less than a minute to produce Alewife machine code on a SPARCstation-I. In fact, for the examples we have run, SDC's run time is dominated by the time taken to compile SDC's output Mul-T code to Alewife machine code.

8 Related Work

Previous techniques developed for compiling dataflow graphs onto a given, possibly parameterized, architecture, do not generally exploit apriori knowledge about the regular, hierarchical structure of the computation [17, 1, 2, 3, 18].

Sarkar's [17] general approach to the multiprocessor compilation problem for programs written in a single assignment language, SISAL, can handle a large class of parameterized architectures, with varying processor and interconnect characteristics. In Sarkar's approach, a dataflow graph is created for the program, with each node representing a collection of operations in the program. Execution profile information is used to estimate node execution times and communication overhead. Then, an explicit graph partitioning of the dataflow graph of the problem determines the tasks for different processors. Finally, either a run-time scheduling system is invoked to automatically schedule the tasks, or a static scheduling of these tasks is determined at compile time. Sarkar used the hierarchical structure of the dataflow graph to simplify some frequently used graph operations, viz, in determining the transitive closure and critical path analysis. However, because partitioning and scheduling is still done on the expanded graph, it is time consuming. Multilevel scheduling and partitioning schemes discussed in our paper are significantly more efficient when the structure of the operators can be characterized by a simple speedup function.

Several recent efforts [1, 2, 3, 18] have focussed on compiling efficiently nested iterative parallel loops, taking the behaviour of the memory hierarchy into account. Nested iterative loops, for example, form the inner code of matrix operators: the matrix product being a triply nested loop. The basic paradigm in these techniques is to minimize communication by choosing compact partitions of the dataflow graph of the loop. The communication is estimated from loop dependencies. In the context of matrix operators, these techniques result in blocking algorithms similar to those presented in Section 4. However, these techniques do not take boundary effects into account. Our work on compiling for an ill-matched number of processors complements their work.

Another important distinction is that the above efforts did not address composition of loops. Each loop in an interdependent loop nest will be partitioned and scheduled on all available P processors, which is the Naive heuristic. We believe that the optimal processor allocation techniques (e.g., Tree and Greedy) of Section 3 form a natural extension of their work.

Polychronopoulos et al. and Wang et al. [4, 5] have developed techniques to compile a possibly imperfectly nested loop by distributing processors among the loops at different nesting levels. However, their algorithms do not attempt to run disjoint, independent subloops in parallel, since they do not analyze the precedences between different portions of the loop nest.

Consequently, all loops at the same nesting level will be run in sequence. The optimal processor allocation techniques (e.g., Tree and Greedy) of Section 3 can be used to yield significant performance gains over running loops in sequence by exploiting interloop parallelism. We are currently exploring a combination of the above methods for optimal processor allocation in general loop nests, that is, using the techniques of Polychronopoulos and others to allocate processors among outer loop nests and our methods for processor allocation between parallel loops at the same nesting level.

9 Conclusion

We have demonstrated the efficacy of a hierarchical compilation strategy for macro dataflow graphs, using matrix expressions as an example. Our hierarchical compilation strategy consists of a processor allocation phase followed by a partitioning phase. In the first phase, the optimal parallelism and sequencing of all macro nodes is determined, using their speedup functions. Then each macro node is optimally partitioned for the specified parallelism, exploiting the regular structure of the computation. The compiler algorithms are computationally efficient.

We implemented several techniques for processor allocation and partitioning in a prototype structure driven compiler, SDC for matrix expressions. Measured speedups on a simulator of the Alewife machine indicated that the Tree technique for processor allocation is best-suited for determining optimal node parallelism and sequencing for tree structured macro dataflow graphs. Even when the graphs are not tree structured, algorithms like Greedy which try to maximize the number of concurrently runnable macro nodes, are far superior to strategies like Naive which run macro nodes one at a time, in sequence. Algorithms based on bin packing were shown to be close to optimal for compiling matrix operators on an arbitrary number of processors. Our techniques can also be applied to the problem of optimally compiling a set of interdependent loop nests, provided speedup functions can be derived for each nest.

10 Acknowledgments

The research reported in this paper is partly funded by DARPA contract # N00014-87-K-0825, NSF grant # MIP-9012773, and by grants from the Sloan foundation and IBM.

References

[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] 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.

[3] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[4] et. al Polychronopoulos, C. D. Utilizing Multidimensional Loop Parallelism on Large-Scale Parallel Processor Systems. *IEEE Transactions on Computers*, 38(9):1285–1296, September 1987.

[5] C. M. Wang and S. D. Wang. Efficient Processor Assignment Algorithms and Loop Transformations for Executing Nested Parallel Loops on Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):71–82, January 1992.

[6] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Workshop on Scalable Shared Memory Multiprocessors*, Kluwer Academic Publishers, 1991. Also appears as MIT/LCS Memo TM-454, 1991.

[7] Allan Gottlieb, B.D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[8] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

[9] G.N.Srinivasa Prasanna and Bruce R. Musicus. Generalised Multiprocessor Scheduling Using Optimal Control. In *Third Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[10] N. Alon and D.J. Kleitman. Covering a Square by Small Perimeter Rectangles. *Discrete and Computational Geometry*, 1:1–7, 1986.

[11] N. Alon and D.J. Kleitman. Partitioning a Rectangle into Small Perimeter Rectangles.

[12] T.Y. Mount D.M. Kong and M. Werman. The decomposition of a square into rectangles of minimal perimeter. *Discrete Applied Mathematics*, 16:239–243, 1987.

[13] T.Y. Mount D.M. Kong and A.W. Roscoe. The decomposition of a rectangle into rectangles of minimal perimeter. *SIAM Journal of Computing*, 1215–1231, 1989.

[14] G.N.Srinivasa Prasanna. *Structure Driven Multiprocessor Compilation of Numeric Problems.* Technical Report MIT/LCS/TR-502, Laboratory for Computer Science, MIT., April 1991.

[15] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implementation*, June 1989.

[16] Kirk Johnson. Semi-C Reference Manual. August 1991. ALEWIFE Memo No. 20, Laboratory for Computer Science, Massachusetts Institute of Technology.

[17] V. Sarkar. *Partitioning and Scheduling Programs for Multiprocessors.* Technical Report CSL-TR-87-328, Computer Systems Laboratory, Stanford University, April 1987.

[18] M. Wolfe. More Iteration Space Tiling. In *Proceedings of Supercomputing '89*, pages 655–664, November 1989.