MIT/LCS/TM-469

COMPILE-TIME TECHNIQUES FOR PROCESSOR ALLOCATION IN MACRO DATAFLOW GRAPHS FOR MULTIPROCESSORS

G. N. Srinivasa Prasanna
Anant Agarwal

June 1992

Compile-time techniques for Processor Allocation in Macro Dataflow Graphs for Multiprocessors

G.N.Srinivasa Prasanna and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
prasanna@masala.lcs.mit.edu
agarwal@mit.edu

Abstract

When compiling a program consisting of multiple nested loops for execution on a multiprocessor, ***processor allocation*** is the problem of determining the number of processors over which to partition each nested loop. This paper presents processor allocation techniques for compiling such programs for multiprocessors with local memory. Programs consisting of multiple loops, where the precedence constraints between the loops is known, can be viewed as macro dataflow graphs. Macro dataflow graphs comprise several macro nodes (or macro operations) that must be executed subject to prespecified precedence constraints. Optimal processor allocation specifies the number of processors computing each macro node and their sequencing to optimize run time. This paper presents computationally efficient techniques for determining the optimal processor allocation using estimated speedup functions of the macro nodes. These ideas have been implemented in a 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, task scheduling, automatic partitioning.

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.

Optimal compilation can be greatly simplified if the computation has a hierarchical structure, that is, if the computation can be represented as a *macro dataflow graph.* A macro dataflow graph is composed of macro nodes, where each macro node has internal structure - viz. is composed of many simple nodes. For example, a matrix expression can be represented as a macro data flow graph, where each macro node corresponds to a basic matrix operator, such as a matrix add or multiply. Programs consisting of multiple nested loops can often be represented as macro dataflow graphs, where each macro node in the dataflow graph corresponds to a nested loop, and where some form of synchronization among the processors executing each 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 (*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. Next, the computations within each macro node are *partitioned* for communication efficiency among the processors assigned to that node. 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. Previous 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.

In this paper, we describe computationally efficient techniques for processor allocation in macro dataflow graphs. We develop algorithms for node parallelism and sequencing, and describe the design of a compiler using these algorithms for matrix expressions, the Structure Driven Compiler, SDC. We also briefly mention techniques for partitioning the macro nodes, since these techniques provide the speedup functions necessary for processor allocation.

Matrix expressions were chosen because their macro dataflow graphs exhibit simple data-independent control, but 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 characterised, enabling speedup functions to be derived by simple analysis. Some examples of matrix expressions are shown below, where all operators are matrix operators. In all that follows, the terms "macro node" and "matrix operator" mean the same.

$$\begin{aligned}
Y &= A(B + CD) \qquad \text{– Simple Matrix Expression} \\
Y &= a_0 + a_1A + a_2A^2 + a_3A^3 + \cdots + a_NA^N \qquad \text{– Matrix Polynomial} \\
Y &= WX \qquad \text{– Fourier Transform, where matrix W: } w_{kl} = e^{-j\frac{2\pi kl}{N}}
\end{aligned}$$

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 (matrix operator) is 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., multiple matrix operations manifested in a program as nested loops), with M macro nodes, the hierarchical compilation strategy speeds up partitioning and scheduling to $\Theta(M)$ 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 (matrix operator). 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.

Next, the dataflow graph of the individual macro nodes (matrix operators) 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.

This paper focusses on the processor allocation phase of the compiler. See [4] for details of the partitioning phase.

1.2 A Simple Example

Consider the following matrix expression (denoted $g1$), whose macro dataflow graph appears in Figure 1(a).

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

Its 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. 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 macro nodes (partially or completely) to yield a large dataflow graph for the expression. Then it partitions and schedules the resulting graph, ignoring pre-existing structure within each macro node, 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 the resulting partition and schedule is globally optimal, this strategy is time consuming.

Figure 1(c) and (d) illustrate the hierarchical compilation strategy. First, in the processor allocation phase, the sequencing and parallelism of all the five macro nodes 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 macro node in the other branch. Finally, the last addition is run on all five processors. At this point, the sequencing and parallelism of every macro node 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 macro node for the number of processors determined above. As depicted in Figure 1(d), the computations in the matrix multiply (node 1) is partitioned into two chunks, M_{10} and M_{11}, for execution on two processors to balance their load and minimize communication.

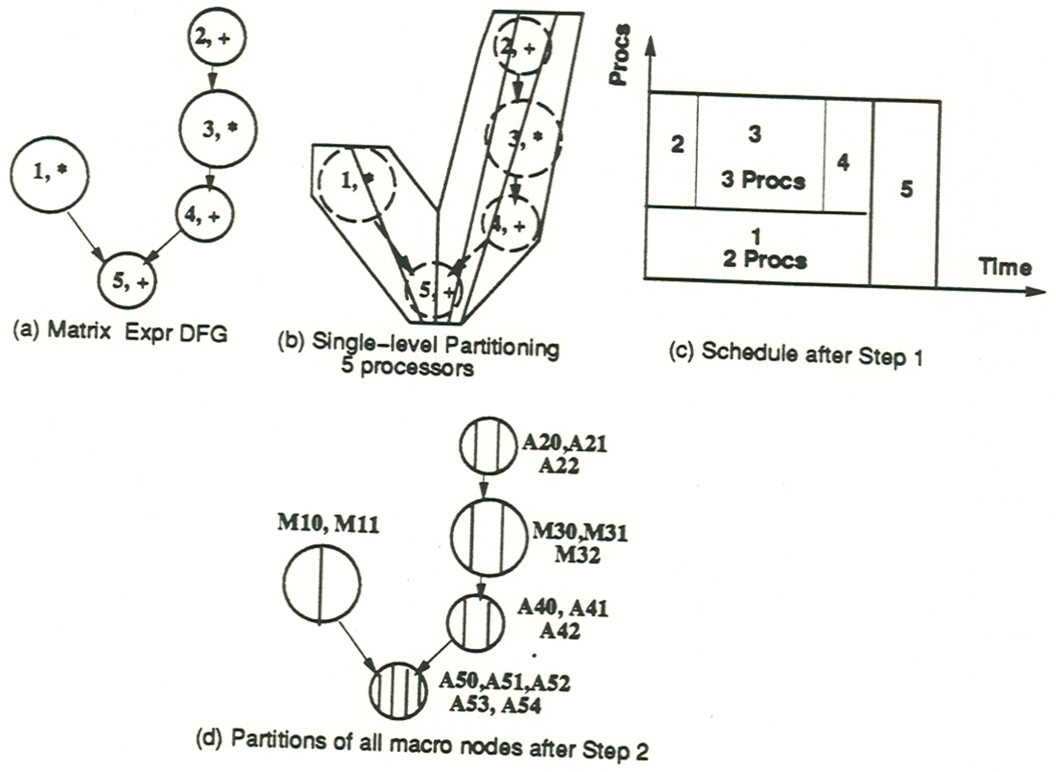

Figure 1: The hierarchical compilation paradigm.

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. Synchronization points are inserted to ensure completion of computation of an macro node 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 in the above example.

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 describes how to determine the processor allocation for each macro macro. Since these techniques depend on speedup functions of macro nodes (matrix operators), Section 4 briefly describes how speedups may be estimated for matrix operators, and how the operators can be partitioned for the level of parallelism determined during processor allocation. Section 5 provides details of our implementation, and Section 6 presents experimental results. Section 7 summarizes related work and Section 8 concludes the paper.

2 Simplifying Assumptions

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

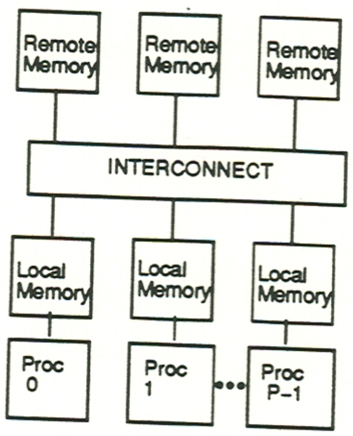

Figure 2: Multiprocessor model.

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 an macro nodes influences the result. Furthermore, the optimal partition and schedule may have portions of an macro node being computed, before all its predecessors have finished (non-strict execution).

SDC makes the following simplifying assumptions. In the first step, it determines the number of processors assigned to an 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 an 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 most 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 accurate if communication within an 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 a simple characterization of the multiprocessor architecture. Our architectural abstraction, depicted in Figure 2, 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. 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).

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.

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 [5]. The fetch-and-operation allows an atomic update of a global datum. For matrix multiplies, for example, the fetch-and-add allows synchronized accumulates to compute each element of the resulting product matrix. The fetch-and-operation can also be used in a software combining tree to implement distributed semaphores [6], which are required to enforce the precedence constraints in the macro data flow 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 [7], 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.

3 Determining Sequencing and Parallelism of Macro Nodes

Two tasks have to be performed in an optimal fashion for processor allocation in a macro dataflow graph: 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.

Finding optimal macro node parallelism and sequencing is a generalization of classical scheduling and will be called *generalised scheduling*. 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 scheduling problem. Then we present a formulation based on optimal control theory, and summarize the results that emerge. We then describe an optimal scheduling technique for tree structured macro dataflow graphs. The section concludes by discussing how the theoretical results are used to derive practical scheduling 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 minimizing overhead. This intuition can be given a rigorous foundation using optimal control theory.

3.2 Control Theoretic Formulation of Scheduling

The fundamental paradigm [8] is to view macro nodes as dynamic systems, whose state represents the amount of computation completed at any point of time. The matrix expression 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 schedule is equivalent to a time-optimal control strategy (optimal processor assignment), and this can be formalised 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) 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 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 [8] can be invoked to yield insights into generalised 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 generalised 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 [8]). 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 allocations are no longer functions of time, but constants. 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 $1 : K$, 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 schedule, S_R, for the reduced graph maps directly into an optimal schedule for the original, S_O as follows. The processor allocations of each macro node in S_O is equal to that of the composite macro node $1 : K$ in S_R, that is

$$P_i(S_O) = P_{1:K}(S_R) \qquad i = 1 \cdots K$$

Therefore, all macro nodes in a series set have the same processor allocation. An 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 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 schedule, S_R for the reduced graph maps directly into an optimal schedule for the original, S_O as follows. All macro nodes in the parallel set are started and finished at the same time. This implies that the processor allocations among the parallel macro nodes is in proportion to the $1/\alpha$ power of their individual lengths, that is,

$$P_i(S_O) = \frac{L_i^{1/\alpha}}{\sum_{j=1}^{K} L_j^{1/\alpha}} P_{1:K}(S_R) \qquad i = 1 \cdots K$$

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 processing schedule for every task in the original graph.

Pseudo code for the Tree Heuristic is shown in Figure 3. 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 subtrees (tree-len), the processor allocation (proc) to this macro node, and the start and stop times of 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 allocations and the start and finish times for all macro nodes. Notice that it attempts to equalize finish times of all predecessors (subtrees) at each macro node, by properly splitting the processor resource amongst them.

```
struct tree {
  op_len;      /* The length of the corresponding macro node*/
  tree_len;    /* The length of the corresponding macro node,
                * plus all subtree lengths */
  proc;        /* Processor allocation for this macro node and
                * all 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 3: Pseudo code for Tree Heuristic.

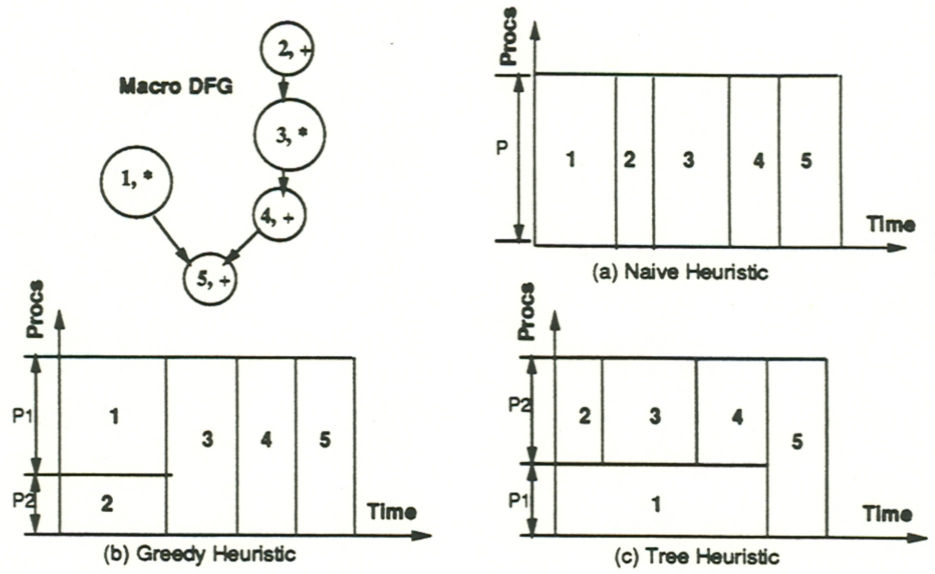

Figure 4: Generalised scheduling heuristics.

The 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 3 (Section 4), is not of 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 6.1). We show that the technique is robust for the value of α used.

3.6 Generalised Scheduling Heuristics

For purposes of comparison, we have incorporated three heuristics - Naive, Greedy, and Tree, into our structure-driven compiler. Figure 4 illustrates the operation of the three heuristics on the tree structured dataflow graph used in Section 1.2. 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 4(a)) runs each macro node on all the available processors. Thus, macro nodes 1, 2, 3, 4, and 5 are run in sequence. Although the execution times with this heuristic are clearly sub-optimal, this heuristic is used for its simplicity.

The Greedy heuristic (Figure 4(b)) is an as-soon-as-possible greedy 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 which 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 allocations (rounded to nearest integers) re

$$P_1 = \frac{L_1^{1/\alpha}}{L_1^{1/\alpha} + L_2^{1/\alpha}} P \qquad \text{and } P_2 = \frac{L_2^{1/\alpha}}{L_1^{1/\alpha} + L_2^{1/\alpha}} P$$

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 allocated 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 allocated 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 4(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 allocated among the subtrees so that their finishing times are equalized. Finally, macro node 5 is run on all the available processors. The processor allocations 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 overviews 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.

The key idea is to exploit the regular structure in the operator dataflow graph: we represent the dataflow graph as a lattice, with each dataflow graph node corresponding to some lattice point. For example, 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 5(a). Each node (i, j, k) represents the computation

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

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. For example, all nodes arranged in a line parallel to the k axis share the same element of A, namely a_{ij}. Nodes arranged in a line parallel to the i axis share b_{jk}. Nodes arranged in a line parallel to the j axis accumulate partial sums to the value of c_{ik}. An optimal partition on P processors divides the operator dataflow graph lattice into

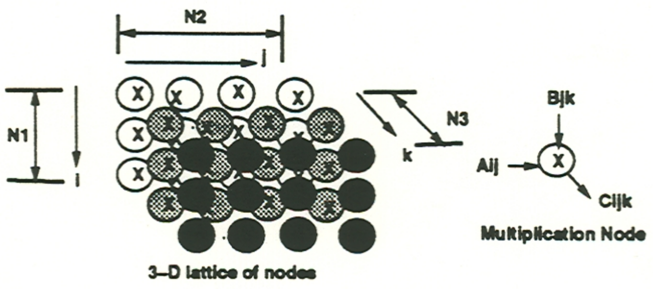

Figure 5: Dataflow graphs for matrix product

P equal sized chunks, while choosing the shape of the chunks in a compact manner to minimise the cost of data accesses (communication) of any chunk.

This lattice representation of the dataflow graph facilitates the partitioning as follows. First, the size of each chunk is its volume V (number of dataflow nodes enclosed). The total computation cost is then simply $(T_a + T_m)V$, since each dataflow graph node is a multiply-add. The communication of any chunk is computed as follows. The number of accesses of elements of A is the projection PA of the chunk on the A (ij) face of the dataflow graph. Similarly, the projections PB and PC on the B and C faces measure the number of accesses of elements of B and accumulates to the output matrix C respectively. Since accesses of A and B are memory fetches (each costing T_u), while those of C are synchronized accumulates (each costing T_{fa}), the total communication cost for this chunk is

$$T_u\,(PA + PB) + T_{fa}PC$$

which is the weighted projected area of the cluster. Since the projected surface area of each chunk has to be minimised keeping their volumes equal, the ideal chunks are cubical boxes (blocks), whose aspect ratio depends on the architectural parameters T_u and T_{fa}. These P ideal chunks have to be packed into the dataflow graph lattice, ideally without distortion. Three-dimensional bin packing techniques can be employed to achieve close to optimal partitions [4].

The processor allocation techniques in Section 3 need the speedup functions $S(P)$ of the optimal operator partitions. These functions are a byproduct of the above analysis. The speedup is the ratio of the execution time on 1 processor to the execution time on P processors. Since the three-dimension bin packing techniques are close to optimal, we can use lower bounds. The lower bound T_e on the execution time, derived by assuming all chunks are equal in size and ideal in shape, and assuming that computation and communication times are additive, can be shown to be:

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

The lower bound on the total time has a linearly decreasing and a less than linearly decreasing component. Denoting $k_1 = (T_a + T_m)$ and $k_2 = 3(T_u^2T_{fa})^{\frac{1}{3}}$, the maximum attainable speedup then becomes,

$$S(P) = \frac{k_1N_1N_2N_3 + k_2(N_1N_2N_3)^{\frac{2}{3}}}{k_1\frac{N_1N_2N_3}{P} + k_2\frac{(N_1N_2N_3)}{P}^{\frac{2}{3}}} \tag{3}$$

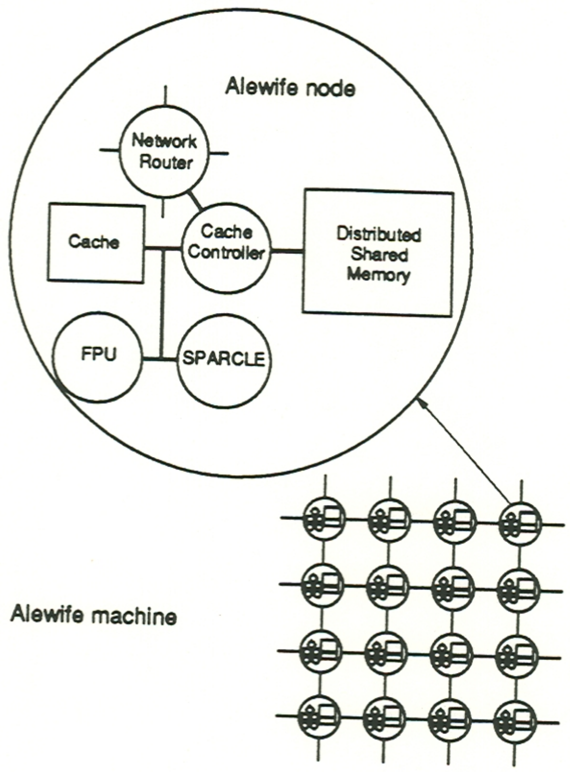

Figure 6: Structure of the Alewife machine.

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.

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 [9], which is compiled and run on the Alewife machine simulator, ASIM. Mul-T is a parallel lisp language.

The Alewife Machine The Alewife Machine [7] is a mesh-connected, distributed shared-memory multiprocessor with coherent caches, as shown in Figure 6. The processor, Sparcle, uses 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. The Sparcle processor uses a load-store architecture, so arithmetic operations are accurately characterised by the execution times (T_a, T_m), ignoring pipeline latency. The local cache in each Alewife node allows fast, 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. 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 first-level store, and a 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 datum 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 tracks 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. 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 constructs, such as distributed semaphores, provided by the Alewife parallel software library. Input and output matrices, and temporaries, are automatically shared among multiple threads through shared memory.

Compiler Implementation The compiler performs a two level partitioning and scheduling assuming that the speedups are of the form P^α. The partitions are grouped into P threads, one for each processor. Code consisting of a sequence of calls to the routines handling the operator partitions 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 be distributed across multiple processors and necessitates a mechanism for keeping track of the constituent chunks. Since data sizes can be arbitrary, and ill-matched to the size of the multiprocessor, the chunks are usually irregular. We provide access to the chunks through pointers duplicated in several processor nodes to avoid hot spots.

6 Experimental results

This section presents experimental results obtained with the 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 $1\times1\times1$ (uniprocessor), $2\times2\times2$, and $4\times4\times4$. 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 first present results for the the matrix product to show how speedup functions are derived. Then using these speedups, we present results for complete expressions. We emphasize that all

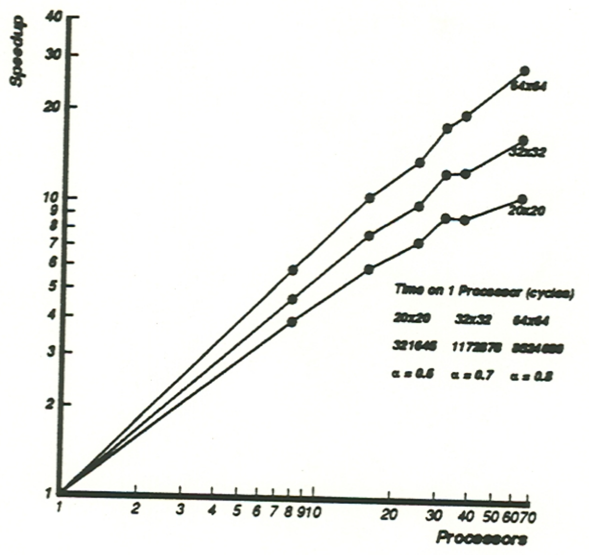

Figure 7: Speedup curves for matrix product.

the speedups shown are close to the "true" speedups for the expression. That is, the uniprocessor program is optimised for a single processor, that for 8 processors is optimised for 8 processors, and so on.

6.1 Matrix Product

Code for a matrix product is generated by the compiler for each number of processors, following the techniques of Section 4. While is is not immediately apparent that the speedup function for matrix multiply shown in Equation 3 is of the form P^α, it can be approximated as such, as is evident from Figure 7. 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 3, which implies that α should lie between 2/3 and 1. We can also estimate α from a knowledge of the multiprocessor constants T_a, T_m, and T_{fa}.

6.2 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, with matrix sizes 32×32.

The speedup curves plotted in Figure 8 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

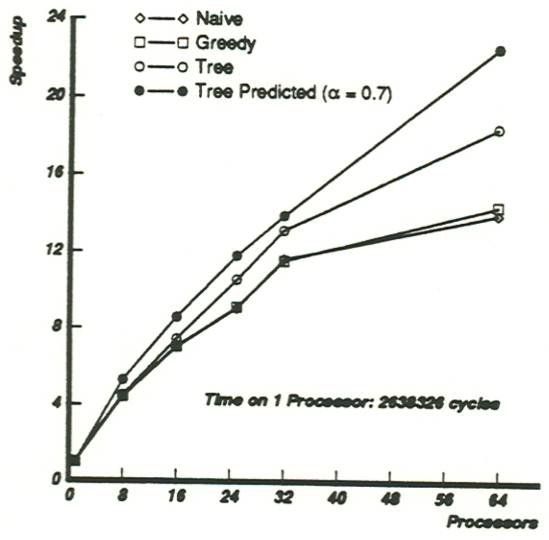

Figure 8: Speedup curves for expression $g1$, with 32×32 matrices.

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

$$S(P) = T(1)/T(P)$$

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

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

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 performance 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 2 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.

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

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

Table 2: Speedup for 64 processors, using various values of α.

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 allocation. After discretization, all the processors get allocated 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 allocation in the first step. The small allocation for the addition causes it to finish late. Consequently, the sensitivity of the processor allocation 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.

6.3 Large Almost Balanced Tree – $g2$

The next example demonstrates that the relative gain in performance from the Tree heuristic increases with increase in the parallelism in the macro dataflow graph. Expression $g2$ shown below is essentially two copies of $g1$ executed in parallel and allows Tree more opportunity to run tasks concurrently.

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

The speedups are plotted in Figure 9. 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.

6.4 Large Unbalanced Product Tree – $g3$

The expression $g3$, is an unbalanced tree, with one branch twice as large as the other.

$$
\begin{array}{l}
(\times\ (\times\ (\times\ A_1\ A_2) \\
\quad\quad (\times\ (\times\ A_3\ A_4)\ A_5)) \\
\quad (\times\ (\times\ A_6\ A_7) \\
\quad\quad (\times\ (\times\ A_8\ A_9) \\
\quad\quad\quad (\times\ (\times\ A_{10}\ A_{11})\ (\times\ (\times\ A_{12}\ A_{13})\ A_{14})))))
\end{array}
$$

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

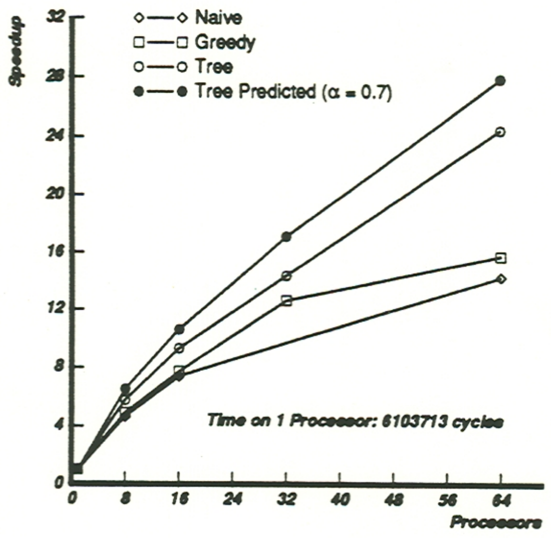

Figure 9: Speedup curves for expression $g2$, with 32×32 matrices.

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 10 shows the speedup curves for the Naive, the Greedy, and the Tree heuristics. The expected speedup for the Tree heuristic has also been plotted. The Tree heuristic performs slightly better than Greedy; both are 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.

6.5 Compilation Time

The compilation time is $\Theta(M)$, where M is the number of matrix operators (macro nodes). By comparision, a compiler using general partitioning and scheduling techniques as in [10] takes time $\Theta(N)$ to $\Theta(N^3)$, where N, the number of simple nodes, is frequently about $100 - 1000$ times the number of processors P, and $N >> M$. 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.

7 Related Work

Previous techniques developed for compiling dataflow graphs onto a given, possibly parameterised, architecture, do not generally exploit apriori knowledge about the regular, hierarchical structure of the computation [1, 2, 10, 3, 11].

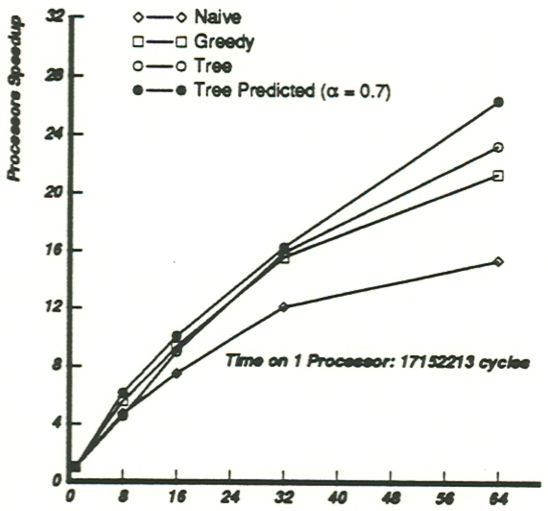

Figure 10: Speedup curves for $g3$, with 32×32 matrices.

Sarkar's [10] general approach to the multiprocessor compilation problem for programs written in a single assignment language, SISAL, can handle a large class of parameterised architectures, with varying processor and interconnect characteristics. In Sarkar's approach, an expanded 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 speed-up function.

Several recent efforts [1, 2, 3, 11] have developed techniques to compile 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 minimise 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, 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) of Section 3 form a natural extension of this work.

8 Conclusion

We have presented computationally efficient techniques for processor allocation in macro dataflow graphs for multiprocessors. Processor allocation is the problem of determining the number of processors allocated to each macro node (or node parallelism) and their sequencing. In our method, the compiler relies on estimated speedup functions for the macro nodes to determine the processor allocation. The operator parallelism is used by subsequent stages of the compiler for partitioning each macro node. Thus, processor allocation followed by techniques to partition each macro node optimally for the specified parallelism forms a two stage hierarchical compilation strategy.

We implemented several techniques for processor allocation and partitioning in a prototype structure drive compiler, SDC, for matrix expressions. Measured speedups on a simulator of the Alewife Machine indicate that the Tree generalised scheduling technique is best-suited for determining operator parallelism and sequencing for tree-structured macro dataflow graphs. 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.

9 Acknowledgments

The research reported in this paper is funded by DARPA contract # N00014-87-K-0825, NSF grant # MIP-9012773, and by grants from the Sloan foundation and IBM. Substantial technical input from Bruce Musicus is also gratefully acknowledged.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] M. Wolfe. Iteration Space Tiling for Memory Hierarchies. 1987.