Minimizing Communication while Preserving Parallelism

Wayne Kelly and William Pugh
Department of Computer Science
University of Maryland, College Park, MD 20742
{wak,pugh}@cs.umd.edu

December 1, 1995

Abstract

To compile programs for message passing architectures and to obtain good performance on NUMA architectures it is necessary to control how computations and data are mapped to processors. Languages such as High-Performance Fortran use data distributions supplied by the programmer and the owner computes rule to specify this. However, the best data and computation decomposition may differ from machine to machine and require substantial expertise to determine. Therefore, automated decomposition is desirable.

All existing methods for automated data/computation decomposition share a common failing: they are very sensitive to the original loop structure of the program. While they find a good decomposition for that loop structure, it may be possible to apply transformations (such as loop interchange and distribution) so that a different decomposition gives even better results. We have developed automatic data/computation decomposition methods that are not sensitive to the original program structure. We can model static and dynamic data decompositions as well as computation decompositions that cannot be represented by data decompositions and the owner computes rule. We make use of both parallel loops and doacross/pipelined loops to exploit parallelism.

We describe an automated translation of the decomposition problem into a weighted graph that incorporates estimates of both parallelism and communication for various candidate computation decompositions. We solve the resulting search problem exactly in a very short time using a new algorithm that has shown to be able to prune away a majority of the vast search space. We assume that the selection of the computation decomposition is followed by a transformation phase that reorders the iterations to best match the selected computation decomposition. Our graph includes constraints to ensure that a reordering transformation giving the predicted parallelism exists.

1 Introduction

The task of mapping a program written in a sequential programming language onto a multi-processor machine can be divided into two subproblems: deciding how to distribute the computation amongst the available processors and deciding how to order the computations. Finding a close to optimal solution in a feasible amount of time for either of these problems in isolation is still an open problem; it is even more difficult to solve these problems simultaneously. To simplify the problem, we solve the sub-problems sequentially (that is, first solve one, and then solve the other based on the solution found for the first). However, in doing so we must be mindful of the fact that the two problems are tightly coupled and use this information when ordering the problems and when devising methods to solve the problems (especially the one we decide to solve first).

In deciding how to distribute the computations amongst the available processors, we want to minimize the amount of communication between processors while at the same time preserving some degree of parallelism. In deciding how to order the computations, we want to minimize the time processors spend waiting for messages from other processors and to access memory locations in an order that exploits the memory hierarchy.

Minimizing communication between processors can be accomplished without regard to the order of the computations. However, achieving sufficient parallelism does depend on the execution order. Through a combination of scalar and array expansion or privatization, loop distribution, statement reordering and loop interchange, it is often possible to expose parallel loops that did not exist in the original program. Even if parallel loops exist in the original program, distributing the iterations of the newly exposed parallel loops rather than the original parallel loops might result in a higher granularity of parallelism or in lower inter-processor communication costs. In some cases, there may be no parallel loops to exploit, but we may be able to transform the program and use doacross/pipelining techniques to allow computation and communication to be overlapped.

We assume that the computation distribution phase will be followed by a transformation phase that will reorder the computations to obtain good performance. So, in evaluating how much parallelism could be achieved by distributing the iterations of a particular loop, we should not be influenced by the original computation order.

One of the things that makes our work difficult is that previous work has not developed techniques that, given a data decomposition, reorders the program so as to exploit parallelism. The only real discussion of this issue is in [HKT91]. While the techniques described there work well for simple stencil computations, they are not theoretically sound and make bad or indecisive decisions in a number of cases, including some realistic cases such as Gaussian elimination kernels [KP95]. Our work includes a model of how to reorder computations so as to exploit parallelism.

Most previous work determines a static data distribution for each array (i.e., a mapping from the elements of that array to virtual processors). Together with the owner-computes rule (where each iteration is performed on the processor that "owns" the array element being written), the distribution specifies a mapping from the iterations of each statement to a set of virtual processors. Virtual processors are folded onto physical processors in a block, cyclic or block-cyclic manner.

We instead determine a *space mapping* for each statement that directly maps each iteration of that statement to a virtual processor. This allows us to represent not only dynamic data distributions, but also non-data distributions, (i.e., we can specify space mappings that cause different iterations of the same statement that write to the same array elements to be executed on different processors). The primary places where non-data decompositions are useful are in the placement of computations that are performing reductions.

However, the main utility of associating a space mapping with each statement is that we can specify *dynamic* data decompositions without having to decompose the program into a sequence of phases between which redistributions are allowed (a process that is heavily dependent on the order of the computations). We can generate dynamic data decompositions by adding constraints to force the computation distribution to be equivalent to a dynamic data distribution.

The problem of automatically distributing computation has been addressed by a large number of authors [Gup92, Fea94, AL93, RKU93, GAL95, SSP$^+$95]. Our work improves on most previous work in the following ways:

1. We are not influenced by the order of the computation in the original program. We use methods to determine the parallelism inherent in the program rather than the parallelism that can be obtained using the computation order in the original program.

2. When analyzing parallelism, we not only examine each loop to determine whether its iterations can be run entirely independently, but also whether its iterations can be pipelined so that computation and communication are overlapped (a lesser but still important form of parallelism). Exploiting this parallelism requires that we use a SPMD rather than a SIMD model.

3. We associate a space mapping with each statement, which allows us to represent dynamic data distributions without having to partition the program into phases, as well as allowing us to represent non-data distributions.

4. We obtain accurate indications of the relative volumes of different inter-processor communications by computing the dimensionality of value-based flow dependence relations[PW93] (an abstraction that precisely describes which iterations actually read values written by which other iterations). Without this, it would be impossible to analyze the communication costs without knowing where an array was going to be redistributed.

5. We solve the search problem exactly in a feasible amount of time by using a number of very effective but safe pruning strategies. Other approaches use heuristic or greedy algorithms.

6. We simultaneously optimize for communication and parallelism, trading one off for the other where necessary to obtain an overall optimal solution.

Throughout this paper, we make a number of simplifying assumptions, such as the assumption that all loops have n iterations. We could eliminate some of these assumptions at the cost of substantial complications to our framework. However, the point of this paper is not to identify which of two decompositions is 10% better than the other; our cost model is not sensitive or accurate enough to answer these kinds of questions. It is unclear if there is any way to answer those kinds of questions other than by performing time trials on the target machine. Our methods are designed to find a distribution such that no significantly better distribution exists, and could be easily altered to generate a list of all such decompositions.

The rest of the paper is organized as follows. In Section 2 we describe our methods to determine the parallelism inherent in the programs we analyze. In Section 3 we describe our communication cost model. In Section 4 we describe our algorithm to simultaneously optimize communication and parallelism, together with the pruning strategies that make it feasible. In Section 5 we describe our alignment algorithm that selects constant offsets to add to the linear space mappings found in Section 4. In Section 6 we give experimental results to demonstrate the efficiency and effectiveness of our algorithms. In Section 7 we discuss related work and finally in Section 8 we state our conclusions.

2 Parallelism Analysis

In this section we describe our methods to determine the parallelism inherent in the programs we analyze. Our first observation is that the most useful form of parallelism is between different iterations of the same statement, rather than between iterations of different statements. This implies that we should examine each statement separately to determine whether any of its iterations can be executed in parallel. In doing so, however, we want to ignore any constraints on parallelism imposed by the original loop order or by other statements in the loop nest. On the other hand, it is clearly not sufficient to examine each statement in isolation. What we need to do is consider all direct and transitive self data dependences of each statement. In previous work [KPRS95] we have described how to compute transitive self dependences using a very precise abstraction called dependence relations. That approach gives very accurate information about parallelism; however, we have found that it can be expensive for very large programs. So, for large programs, we have developed a more efficient but potentially less accurate approach to computing transitive self dependences.

We first perform data dependence analysis to produce a set of extended dependence direction vectors[WB87, Wol91] between each pair of statements. A dependence direction vector is a vector $(v_1, \ldots, v_m)$ where v_i is either '-', '0', or '+', indicating whether the difference between the level i index variable at the source of the dependence is less than, equal to, or greater than the level i index variable at the sink of the dependence. For a normal direction vector, the length is equal to the maximum common loop depth of the two statements; an extended direction vector uses the minimum loop depth of the two statements. For example, the dependence from statement 1 to statement 2 in Figure 1 below is represented by the direction vector (0) and by the extended direction vector $(0, -1)$.

```
     for i = 1 to n
         for j = 1 to n
1:           a(i,j) = ...
         for j = 1 to n
2:           ...  = a(i,j+1)
```

Figure 1: Example to illustrate extended direction vectors

In performing this analysis, we ignore reduction dependences (dependences between two updates to the same memory location), since given certain reasonable assumptions they do not substantially inhibit parallelism.

We use a modified form of the Floyd-Warshall algorithm to compute transitive dependences (see Figure 2). The input to this algorithm is a set of variables d_{ij} representing all direct data dependences from statement i to statement j. The output is a set of variables t_{ij} representing all transitive data dependences from statement i to statement j. When taking the union of two sets of extended direction vectors, we combine vectors if and only if it will not lead to the loss of information. For example, we can combine $(0, +)$ and $(0, 0)$ to produce $(0, 0+)$; however, we can't combine $(0, +)$ and $(+, 0)$ to produce $(0+, 0+)$ (as that would imply that $(0, 0)$ and $(+, +)$ are possible direction vectors). When taking the union of sets of extended direction vectors with different lengths (which occurs when considering transitive dependences through statements which are not as deeply nested as the statement in question), we pad the shorter vectors with *'s, indicating that '-', '0' and '+' are all possible. The composition of direction vectors is performed element-wise and is defined in the obvious manner (i.e. composing '+' and '0' produces '+', composing '+' and '-' produces '*', etc).

```
for i = 1 to n
    for j = 1 to n
        t_ij = d_ij
for k = 1 to n
    for i = 1 to n    if (i ≠ k)
        for j = 1 to n    if (j ≠ k)
            t_ij = t_ij ∪ (t_ik ∘ t_kk ∘ t_kj)
```

Figure 2: Modified form of Floyd-Warshall algorithm to compute transitive closure

In our current implementation, the set of candidate space mappings consists of each dimension in the original iteration space plus the zero (which corresponds to not distributing). We wish to analyze the parallelism that would result from selecting each of these candidate space mappings without being influenced by the original loop order. We consider all legal loop permutations of the statement (including all combinations of reversing the loops) and classify the candidate space mappings according to the amount of synchronization they will require (and hence how much parallelism they permit) within each particular permutation. Even if we have to consider all permutations of the loops (which is exponential in the number of loops), we can do so in a small amount of time[1] since we consider each statement separately and statements are seldom nested within more than 4 or 5 loops. Each candidate space mapping is given an overall classification based on the amount of synchronization it will require using the best permutation for that particular candidate.

2.1 Synchronization costs

To analyze the amount of synchronization that will be required if we use a particular candidate with a particular permutation, we need to consider the structure of the loops that would ultimately be used. We analyze the general case of using a block-cyclic distribution rather than separately analyzing both the block and cyclic cases. If the candidate loop is at level y in the current permutation then a straight forward implementation of block-cyclic distribution would lead to the following loop structure:

for t_1
 ...
  for t_{y-1}
   for t_y^B
    for t_y^P
     for t_y
      ...
       for t_m
        stmt

The t_y^P loop iterates over the set of physical processors, the t_y^B loop iterates over the blocks and the t_y loop iterates over the iterations within each block. In a block distribution the t_y^B loop will be degenerate and in a cyclic distribution the t_y loop will be degenerate. If x is the deepest loop level that carries a transitive self-dependence with a negative dependence distance in the distributed loop then it is legal to move the t_y^B loop out to just inside the t_x loop:

for t_1
 ...
  for t_x
   for t_y^B
    for t_{x+1}
     ...
      for t_{y-1}
       for t_y^P
        for t_y
         ...
          for t_m
           stmt

To convert this to SPMD code we would:

[1] Although we have a number of ideas for more efficient algorithms for doing this analysis, this analysis step has not required significant time in any of our experiments.

- Replace the t_y^P loop with a conditional statement placed outside the t_1 loop.
- Insert a barrier inside the t_x loop. This will enforce all dependences carried by loops t_1 through t_x.
- Insert post-and-wait style synchronization to enforce any dependences carried by loops t_y^B through t_{y-1}.

By moving the t_y^B loop out as far as possible, we execute a minimal number of barriers. The placement of the t_y^B loop also implies that any dependences carried by loops t_{x+1} through t_{y-1} will be from a lower numbered physical processor to a higher numbered physical processor, so some form of parallelism (either pure or pipelined) will result within each iteration of the t_y^B loop.

Dependences carried by the t_y^B loop may go either up or down in physical processor number and so, going from one iteration of the t_y^B loop to the next, may cause the pipeline to be interrupted. Sub-section 2.1.1 explains how we estimate whether or not the pipeline will be interrupted. If we estimate that the pipeline will be interrupted, then we pessimistically assume that the post-and-wait synchronization inserted for dependences carried by the t_y^B loop has the same effect as a barrier placed inside the t_y^B loop. Otherwise, we will not have to wait on these dependences and can ignore them in our synchronization cost estimates. So, the number of barrier synchronizations that we perform will be either n^x or $\frac{n^{x+1}}{BP}$, where n is the number of iterations per loop, B is the block size and P is the number of physical processors.

We compute D, the maximum amount by which different processors will be out of lock-step, as follows:

- If any dependences are carried by the distributed loop, we expect a delay of $L + Bn^{m-y}$ between the time processor p can start and the time processor $p+1$ can start, where L is the inter-processor message latency. The wait from when the first processor reaches a barrier until the last reaches the barrier will be $P-1$ times the delay between successive processors. We simplify this slightly to estimate $D = P(L + Bn^{m-y})$.
- If no dependences are carried by the distributed loop, but there are inter-processor dependences carried by loops t_{x+1} through t_{y-1}, then those dependences from processor p to processor $p+1$ may force processor $p+1$ to lag L behind processor p. We again simplify slightly and estimate $D = PL$.
- If no inter-processor dependences are carried by loops t_{x+1} through t_y, then the processors should remain synchronized to within $D = L$.

To perform a barrier synchronization, the processors must exchange messages (costing L) and synchronize (costing D). Since $D \geq L$, we simplify the barrier cost to D.

2.1.1 Pipeline interruption

Consider a dependence carried by the t_y^B loop from the last processor to the first processor. As we estimated before, the last processor may be lag behind the first processor by up to D time. We may therefore have to wait $D + L$ units of time between when a message is sent by a statement instance in iteration b_1 of the t_y^B loop on processor p and when a corresponding message is received by a statement instance in iteration $b_1 + 1$ of the t_y^B loop on processor p.

However, any dependences carried by the t_y^B loop will be either forward or loop independent with respect to loops t_{x+1} through t_{y-1}. So, during this time, processor p will be able to execute all of block b_1 which will require Bn^{m-x-1} units of times. We therefore predict that the pipeline will be interrupted if and only if $D + L > Bn^{m-x-1}$.

2.2 Load balance

We examine the loop bounds of each statement to determine whether the amount of work in each iteration will be constant. If any statements have unbalanced loops, then in addition to considering a block distribution, we also consider a cyclic distribution. When evaluating block distributions for candidates with unbalanced workloads, we add an additional $n^{m-1}/2$ time to our overhead estimate (intended to represent the difference between the amount of work in a rectangular iteration space and a triangular iteration space). In Section 3 we will see how the communication estimates will be higher for cyclic distributions.

2.3 Compatible candidates

After parallelism analysis has been performed, we will know the minimum degree of synchronization that will be required for each candidate. For each candidate there is a set of legal loop permutations that lead to this minimum degree of synchronization. For example, the candidate space mapping $\{[k,i] \to [k]\}$ for statement 1 in Figure 3 will produce parallel execution at loop depth 2 only if the following legal loop permutation is used for statement 1: $\{(i,k)\}$. Similarly, the candidate space mapping $\{[k,i,j] \to [j]\}$ for statement 2 will produce parallel execution at loop depth 2 if any of the following legal loop permutations are used for statement 2: $\{(k,-j,-i),(k,-j,i),(k,j,-i),(k,j,i)\}$.

```
    for k = 1 to n
        for i = k+1 to n
1           a(i,k) = a(i,k) / a(k,k)
            for j = k+1 to i
2               a(i,j) = a(i,j) - a(k,j)*a(i,k)
```

Figure 3: Gaussian elimination

Unfortunately, in this case, because of data dependences, the first statement's permutation can not be used with any of the second statement's permutations. In other words, if we select candidate $\{[k,i] \to [k]\}$ for the first statement and candidate $\{[i,j,k] \to [j]\}$ for the second statement, then we will not be able to achieve parallelism at loop depth 2 for both statements no matter how we reorder the iterations. Thus, analyzing parallelism for each statement in isolation can lead to overestimation of parallelism.

To address this problem, we consider all pairs of statements (p,q) and determine which candidates of statement p

are compatible with which candidates of statement q. Candidate c_p of statement p is compatible with candidate c_q of statement q if there exists permutations π_p and π_q for statements p and q respectively such that π_p requires the minimum degree of synchronization to be used for c_p, π_q requires the minimum degree of synchronization to be used for c_q and π_p is compatible with π_q (see Subsection 2.4).

This compatibility information will be incorporated into the graph constructed in Section 4 to try to ensure that parallelism is not overestimated in the way described above. The compatibility tests we have described are only performed on each pair of statements in isolation. It is theoretically possible for each pair of selected candidates to be compatible but for the set of candidates as a whole to be incompatible. However, since we use transitive dependences in determining which permutations are compatible (see Subsection 2.4), it is very unlikely that this will occur. We have analyzed all of the benchmarks used in this paper and determined in all cases that there do exist globally compatible permutations for all statements that produce the degrees of parallelism as estimated by our system.

2.4 Compatible permutations

To determine whether permutation π_p for statement p is compatible with permutation π_q for statement q we use the following test. First, we construct a set of direction vectors that describe to the order in which the iterations of statement q will be executed if we apply loop permutation π_q. These direction vectors don't correspond to actual data dependences, but rather to ordering constraints that will be satisfied if that permutation is used. For example, if we were going to use permutation (k, i, j) for statement 2 in Figure 3 then we would construct the following set of direction vectors $\{(0, 0, +), (0, +, *), (+, *, *)\}$. In general, the set will be:

$$c_{qq} = \bigcup_{m \in \{0,\ldots,n-1\}} \left\{ \pi_q(\underbrace{0,\ldots,0}_{m}, +, \underbrace{*,\ldots,*}_{n-m-1}) \right\}$$

where $\pi_q(x_1, \ldots, x_n)$ means apply permutation π_q to the vector $(x_1, \ldots, x_n)$. Applying a permutation to a direction vector also involves reversing directions as indicated by the permutation. For example, if we were going to use permutation $(k, j, -i)$ for statement 2 then we would construct the following set of direction vectors $\{(0, +, 0), (0, *, -), (+, *, *)\}$.

Next, we combine these ordering constraints with the transitive dependences between statements p and q to infer new ordering constraints on statement p under the assumption that we will use permutation π_q for statement q. The new ordering constraints are:

$$c_{pp} = t_{pq} \circ c_{qq} \circ t_{qp}$$

Permutation π_p is compatible with permutation π_q if and only if π_p is legal with respect to the new set of ordering constraints c_{pp}. For example, if in the example from Figure 3, $\pi_2 = (k, j, -i)$ then:

$$\begin{aligned} c_{22} &= \{(0, +, 0), (0, *, -), (+, *, *)\} \\ t_{12} &= (0, 0) \\ t_{21} &= (+, 0+) \\ c_{11} &= (+, *) \end{aligned}$$

So permutation (i, k) for statement 1 is not compatible with permutation $(k, j, -i)$ for statement 2 because (i, k) is not legal with respect to c_{11}.

3 Communication Analysis

Now that we have analyzed the parallelism available in each statement, we estimate the communication incurred by each distribution. Communication analysis requires a finite set of candidate space mappings for each statement. In this section we only consider linear space mappings. In Section 5 we explain how to select constant offsets to add to the linear mappings.

Our primary assumption is that communication will only be required between processors if one processor writes a value to a location and some other processor later reads that value from that location. We use value-based flow dependence relations [PW93] to obtain accurate indications of the relative volumes of different inter-processor communications. Value based dependence relations precisely describes which iterations actually read values written by which other iterations. For example, there is no value based flow dependence from statement 1 to statement 3 in Figure 4, since all memory locations written by statement 1 are overwritten by statement 2 before statement 3 can read them. The value based flow dependence from statement 2 to statement 3 would be represented by the dependence relation $\{[i, i] \to [i, i] \mid 1 \le i \le n\}$. From this information we can determine that only n values will be communicated from statement 2 to statement 3, despite the fact that both statements have n^2 iterations.

```
    for i = 1 to n
        for j = 1 to n
1:          a(i,j) = ...
2:          a(i,j) = a(i,j) + ...
3:          ...  = a(i,i) + ...
```

Figure 4: Value based dependence example

We simplify matters by assuming that all loops have some unknown constant number of iterations "n" (even those with known constant loop bounds). This allows us to associate with each value based flow dependence a dimensionality (or rank). This, in turn, allows us to obtain accurate indications of the relative volumes of different inter-processor communications without having to resort to complex and expensive symbolic volume estimation algorithms [Pug94].

For each value-based flow dependence, we consider each combination of candidate space mappings for the statements involved in the dependence. By examining the equality constraints in the dependence relations we try to determine the difference between the virtual processor to which the first

```
        for t = 0 to ITERS
            for j = 0 , DIM-1
                for k = 1, DIM-1
1:                  X[j, k] = ...
            for j = 0 , DIM-1
2:              ...  = X[j, DIM-1] + ...
```

$$d_{12} : \{[t,j,k] \rightarrow [t,j] \mid k = \mathrm{DIM} - 1 \wedge 0 \leq t \leq \mathtt{ITERS}$$
$$\wedge 0 \leq j < \mathtt{DIM} \wedge 2 \leq \mathtt{DIM}\}$$

Figure 5: Value based dependence example

		Stmt 2		
		0	t	j
Stmt 1	0	z	nc	nc
	t	nc	z	nc
	j	nc	nc	z
	k	DIM-1	nc	nc

z : zero, nc : not constant

Table 1: Virtual processor differences

statement is mapped and the virtual processor to which the second statement is mapped. We check for possible differences of: zero , a constant other than zero, and not constant. Table 1 shows each combination of space mappings for statements 1 and 2 in Figure 5, with information about the difference in virtual processors for the value based flow dependence shown.

If the difference in virtual processors is zero then we estimate that there will be no communication. In the case of self dependences, this estimate is always exact. In the case of non self dependences, this constitutes an optimistic assumption that the statements will be assigned identical constants in Section 5. If the difference in virtual processors is a non-zero constant, then we assume that *nearest neighbor communication* will occur. Again, in the case of self dependences, this is always true. In the case of non self dependences, this constitutes an pessimistic assumption that the difference between the constants assigned to the statements will not be the same as the non-zero constant difference between the virtual processors (if that were the case then there would be no communication).

If we are considering a blocked distribution and the communication is nearest neighbor, then many of the dependences will be between different virtual processors that are mapped to the same physical processor. So, for a dependence of dimension n^d, we estimate the amount of inter-processor communication as n^d/B (where B is the number of virtual processors in each block). If the communication is not nearest neighbor, or if we are considering a cyclic distribution, then a dependence with dimensionality n^d has a communication estimate of n^d.

4 The Search Problem

We now represent the space mapping selection problem as a weighted graph. The graph will contain a node corresponding to a each candidate space mapping of each statement. The node weights will be the parallelism overheads as derived in Section 2 and the edge weights will be the communication estimates as derived in Section 3.

4.1 Incompatible candidates

If incompatible candidates exist (see Sub-section 2.4), the parallelism that can be achieved using some candidate space mapping may depend on which candidates are chosen for other statements. We therefore use two nodes to represent such candidates. In one node, we will optimistically assume that the degree of parallelism that we estimated when considering the statement in isolation can be achieved and in the other node will pessimistically assume that choices made for other statements will force us to use a permutation that leads to no parallelism for this statement. The communication costs will be the same for both nodes. If two candidates are incompatible then we do not want to simultaneously select the optimistic version of both candidates. To ensure that does not occur, we add an edge with infinite cost between the optimistic versions of incompatible candidates.

4.2 The simple search procedure

Many previous approaches to automatically minimizing interprocessor communication use heuristic or greedy algorithms. We instead solve the problem exactly. The problem is to select exactly one node from the weighted graph for each statement such that the sum of the node costs of selected nodes and sum of edge costs between selected nodes is minimized. Our basic approach is simple; we perform an exhaustive search through all possible selections of nodes and choose the one with the lowest overall cost. In order to solve the problem in a feasible amount of time, we have developed a number of effective but optimality-preserving pruning strategies. Figure 6 shows a simplified version (without any optimizations) of the recursive depth-first search algorithm. C_k is the list of candidate space mappings for statement k, N is the number of statements, $v_i(s)$ is the parallelism overhead associated with using mapping s for statement i, and $e_{ij}(s_i, s_j)$ is the total cost of communication between statements i and j under mappings s_i and s_j respectively (or infinity if the nodes correspond to the optimistic versions of incompatible candidates).

4.3 Pruning Strategies

Once we have determined the cost of at least one solution, we can use that cost to prune our search space. Suppose we have a solution with cost c and we are considering a partial solution defined by a function S from some subset of the statements to the space mappings currently being considered for those statements. We define:

$$P(S) = \sum_{i \in domain(S)} v_i(S[i]) + \sum_{j \in domain(S) \wedge j \leq i} e_{ij}(S[i], S[j])$$

```
search (statement k)
    foreach space mapping s ∈ C_k
        S[k] = s
        if (k < N)
            search(k+1)
        else
            cost = Σ_{i=1}^{N} (v_i(S[i]) + Σ_{j=1}^{i} e_ij(S[i], S[j]))
            if (cost < best_cost)
                best_cost = cost
                record {S[1],...,S[N]} as best

Start by calling search(1)
```

Figure 6: Simplified search algorithm

If $P(S) \geq c$, then we can terminate consideration of the partial solution, since its total cost cannot be better than that of the solution we already have.

We can perform further pruning if we can determine a lower bound on the cost that will be contributed by statements for which a candidate has not yet been chosen (i.e., without actually considering all combinations of selections and choosing the best one). The lower bound that we use is:

$$lb(S) = \sum_{i \notin domain(S)} \min_{s \in C_i} (P(S \cup \{i \rightarrow s\}) - P(S))$$

That is, $lb(S)$ is the sum of the edge costs from all statements for which a candidate has been chosen to the best candidates of each of the statements for which a candidate has not been chosen, plus the node costs of each of these candidates. If $P(S) + lb(S) \geq c$ then we can terminate consideration of the partial solution.

Given that complete solutions with low costs allow us to prune more than complete solutions with high costs, it is advantageous to find low cost solutions early in the search process. We are free to consider the candidate space mappings for each statement in any order, so we choose an order that is most likely to lead to a complete low cost solution as early as possible. For each unselected statement i, we order the candidates space mappings s according to $P(S \cup \{i \rightarrow s\})$.

We are also free to consider the statements in any order. We will usually have to extensively explore the subtree of the best candidate for each statement, however we would like to avoid having to explore the subtrees of the other candidates. If selecting the second-best candidate of a statement will cause the total cost to rise substantially then we may only have to choose mappings for a few more statements (if any) before the total cost rises to a point where we can prune away the partial solution. So, when selecting the statement to explore next, we choose the one whose second best candidate will add the most to the total cost; that is, the statement i whose second best candidate s is most expensive according to:

$$P(S \cup \{i \rightarrow s\}) + lb(S \cup \{i \rightarrow s\})$$

Note that there is no fixed order in which the statements are considered. At each stage, the statement that we consider next will depends on the current context.

According to the above formula, deciding which statement to consider next would take approximately $O(N^4 M^2)$ time, where M is the average number of candidates per statement. By storing partial sums we can reduce this to $O(NFM^2)$, where F is the average number of statements that have value-based flow dependences reaching a statement. While this is still relatively expensive, it more than pays for itself my substantially increasing the number of pruning opportunities. In our experiments we have not found the cost prohibitive; in Section 6 we present results from experiments showing the efficiency of our pruning strategy.

The above expressions simply add together parallelism overheads and communication costs. In practice, we first multiply the parallelism overheads by a machine dependent constant that represents the ratio of computation speed to communication speed. By varying this parameter, we can determine whether or not a given solution is likely to be optimal across a wide variety of machines.

5 Alignment

Adding a constant to the space mappings can eliminate some nearest-neighbor communications. A constant offset will have no affect on communication between different iterations of the same statement. For dependences between different statements, however, if the dependent iterations map to virtual processors separated by a constant distance, then adding appropriate constants to the selected space mappings, can map them to the same virtual processors. We consider adding constants that are affine functions of the symbolic constants in the program. For example, for the program in Figure 1, we might select "DIM-1" as the constant to add to the space mapping of statement 1.

We use a greedy algorithm to decide which dependences with constant virtual processor distances will be made intra-processor. We maintain a partitioning of the statements such that within each partition, we know the relative differences between their constant parts. Initially all statements are in separate partitions. We process the dependences in decreasing order based on their dimensionalities. For a dependence from statement p to statement q where statements p and q are in different partitions, we can make the dependence intra-processor. In this case, we merge the partitions containing statements p and q and record the virtual processor difference between statements p and q. When we have processed all dependences or when we have only one partition, we arbitrarily choose a constant for one statement in each partition and use it to compute constants for the other statements in the partitions.

6 Experimental Results

We first give experimental results to show that our algorithms execute in a feasible amount of time. The left half of Table 2 gives a breakdown of our execution times for a variety of benchmark programs[2] The times listed are in seconds and are

[2] These programs together with the results that we obtain for them are available in ftp://ftp.cs.umd.edu/pub/omega/results_KP95

								Search time			
								Unoptimized		Optimized	
Program	Nr Stmt	Max Nest	Parallelism Analysis	Comm. Analysis	Search	Align	Total	Calls	Time	Calls	Time
ge	2	3	0.04	0.02	0.00	0.00	0.08	4	0.00	2	0.00
ch	3	3	0.05	0.02	0.00	0.01	0.11	9	0.00	2	0.00
relax	1	3	0.06	0.02	0.00	0.00	0.09	1	0.00	1	0.00
jacobi	3	3	0.10	0.03	0.00	0.01	0.16	6	0.00	3	0.00
across	4	1	0.00	0.01	0.00	0.03	0.03	15	0.00	4	0.00
gosser	5	3	0.08	0.03	0.00	0.10	0.16	112	0.00	5	0.00
choles	6	3	0.03	0.03	0.00	0.01	0.11	201	0.01	6	0.00
burg2	11	2	0.14	0.06	0.01	0.02	0.32	17839	0.99	11	0.01
lczos	23	3	0.14	0.10	0.02	0.06	0.40	7132375	563.93	23	0.02
givens	9	3	0.20	0.08	0.02	0.02	0.41	20776	1.28	10	0.02
cholsky	14	4	0.95	0.18	0.02	0.04	1.34	458656	69.04	14	0.02
intbal	41	2	0.14	0.17	0.03	0.08	0.58	71120230	$\sim$ 1 hour	41	0.03
eflux	27	3	0.43	0.19	0.04	0.08	0.90	2.9×10^{11}	$\sim$ 6 months	27	0.04
mxm	2	3	0.01	0.00	0.00	0.00	0.02	4	0.00	2	0.02
nas3	9	4	0.78	0.14	0.01	0.03	1.06	75926	5.86	9	0.01
emit	34	3	1.55	0.48	0.05	0.08	2.68	1.3×10^{12}	$\sim$ 2 years	34	0.05
vpenta	53	2	0.82	0.77	0.17	0.18	2.36	3.7×10^{16}	$\sim 10^5$ years	53	0.17
adi	17	3	1.56	0.32	0.02	0.05	2.28	2.3×10^{7}	$\sim$ 20 minutes	17	0.02
shallow	65	2	0.38	0.49	0.18	0.15	1.54	4.6×10^{14}	$\sim$ 700 years	90	0.18
erle	60	3	0.82	0.76	0.30	0.18	2.56	3.6×10^{21}	$\sim 10^{10}$ years	105	0.30

Table 2: Breakdown of compilation times for various benchmark programs

as measured by Quantify[3] on a SPARCstation 10/51.

The right half of Table 2 shows a comparison between the unoptimized version of our search algorithm (Figure 6) versus our fully optimized pruning search algorithm. The Calls column is the number of recursive calls to the search procedure , while the Time column is amount of time in seconds, spent in the search procedure (times marked with an $\sim$ are projected times).

6.1 Effectiveness

The first program that we use to demonstrate our effectiveness is adi (as shown in Figure 7), a program fragment used in alternating direction implicit integration. Our parallelism analysis phase produces the results shown in Table 3. If we set our computation to communication ratio parameter to a low value such as 0.01, then we obtain the space mappings shown in Figure 8(a). These space mappings result in all statements being executed in parallel; however, they also result in 4 n^3 and 12 n^2 inter-processor communications. If we set our computation to communication ratio parameter to a more realistic value such as 1.0, then we obtain the space mappings shown in Figure 8(b). These space mappings result in the first four statements being parallelized and three of the last four statements being pipelined, but they result in only 4 n^3 and 4 n^2 inter-processor communications. Other researchers[AAL95] have shown that data distributions analogous to these space mappings produce close to linear speedups on several shared memory machines. If we set our computation to communication ratio parameter to an even higher value such as 100.0 (as might be the case for a network of workstations), then we obtain the space mappings shown in Figure 8(c). These space mappings result in all statements being executed sequentially, with no inter-processor communication.

```
   for t = 0 to ITERS
     for j = 0 to DIM-1
       for k = 1 to DIM-1
1        X[j,k] = X[j,k]-X[j,k-1]*A[j,k]/B[j,k-1]
2        B[j,k] = B[j,k]-A[j,k]*A[j,k]/B[j,k-1]
     for j = 0 to DIM-1
3      X[j,DIM-1] = X[j,DIM-1]/B[j,DIM-1]
     for j = 0 to DIM-1
       for k = DIM-2 to 0 by -1
4        X[j,k] = (X[j,k]-A[j,k+1]*X[j,k+1])/B[j,k]
     for j = 1 to DIM-1
       for k = 0 to DIM-1
5        X[j,k] = X[j,k]-X[j-1,k]*A[j,k]/B[j-1,k]
6        B[j,k] = B[j,k]-A[j,k]*A[j,k]/B[j-1,k]
     for k = 0 to DIM
7      X[DIM-1,k] = X[DIM-1,k]/B[DIM-1,k]
     for j = DIM-2 to 0 by -1
       for k = 0 to -1+DIM
8        X[j,k] = (X[j,k]-A[j+1,k]*X[j+1,k])/B[j,k]
```

Figure 7: Example program: adi

[3] Registered trademark of Pure Software Inc.

Stmt	Transitive	Candidate space mappings					
	Dependences	t		j		k	
1	(+,*,*) (0,0,+)	n^3	(sequential)	nL	(parallel depth 2)	$nP(L+B)$	(pipeline depth 2)
2	(+,*,*) (0,0,+)	n^3	(sequential)	nL	(parallel depth 2)	$nP(L+B)$	(pipeline depth 2)
3	(+,*)	n^2	(sequential)	nL	(parallel depth 2)		
4	(+,*,*) (0,0,+)	n^3	(sequential)	nL	(parallel depth 2)	$nP(L+B)$	(pipeline depth 2)
5	(+,*,*) (0,+,0)	n^3	(sequential)	$nP(L+B)$	(pipeline depth 2)	nL	(parallel depth 2)
6	(+,*,*) (0,+,0)	n^3	(sequential)	$nP(L+B)$	(pipeline depth 2)	nL	(parallel depth 2)
7	(+,*)	n^2	(sequential)			nL	(parallel depth 2)
8	(+,*,*) (0,+,0)	n^3	(sequential)	$nP(L+B)$	(pipeline depth 2)	nL	(parallel depth 2)

Table 3: Synchronization costs for adi, assuming $P(L+B) \leq Bn$

$1 : \{[t,j,k] \rightarrow [j]\}$	$1 : \{[t,j,k] \rightarrow [j]\}$	$1 : \{[t,j,k] \rightarrow [0]\}$
$2 : \{[t,j,k] \rightarrow [j]\}$	$2 : \{[t,j,k] \rightarrow [j]\}$	$2 : \{[t,j,k] \rightarrow [0]\}$
$3 : \{[t,j] \rightarrow [j]\}$	$3 : \{[t,j] \rightarrow [j]\}$	$3 : \{[t,j] \rightarrow [0]\}$
$4 : \{[t,j,k] \rightarrow [j]\}$	$4 : \{[t,j,k] \rightarrow [j]\}$	$4 : \{[t,j,k] \rightarrow [0]\}$
$5 : \{[t,j,k] \rightarrow [k]\}$	$5 : \{[t,j,k] \rightarrow [j]\}$	$5 : \{[t,j,k] \rightarrow [0]\}$
$6 : \{[t,j,k] \rightarrow [k]\}$	$6 : \{[t,j,k] \rightarrow [j]\}$	$6 : \{[t,j,k] \rightarrow [0]\}$
$7 : \{[t,k] \rightarrow [k]\}$	$7 : \{[t,k] \rightarrow [DIM-1]\}$	$7 : \{[t,k] \rightarrow [0]\}$
$8 : \{[t,j,k] \rightarrow [k]\}$	$8 : \{[t,j,k] \rightarrow [j]\}$	$8 : \{[t,j,k] \rightarrow [0]\}$
$(a) : r = 0.01$	$(b) : r = 1.0$	$(c) : r = 100.0$

Figure 8: Selected space mappings for adi

Related work	1	2	3	4	5	6
[Gup92]		√				√
[AL93]		√				√
[RKU93]		√			√	√
[Fea94]			√	√		
[GAL95]					√	√
[SSP+95]			(a)	√	√	
Our system	√	√	√	√	√	√

(a) - Starts with a functional language, so not relevant

Table 4: Properties (from Section 1) of related work

Many papers have been written about automatic data decomposition and most contain examples to show the performance of their respective algorithms. Whilst the papers themselves contain impressive results (and our algorithm/implementation has derived those same data distributions), we have found that in the few implementations that we have been able to experiment with, most of these algorithms are very fragile. That is, the programs as given in these papers can be compiled very efficiently, but minor, semantic-preserving changes to these programs (such as performing loop interchange, loop fusion or statement reordering), often result in completely different and often far from optimal distributions.

Our aim is a system that produces the same result (hopefully an optimal result) regardless of the form in which the program is originally presented. In Figure 9, we demonstrate this aspect of our system by showing the results of applying our system to all six legal loop permutations for Cholesky decomposition. We have not found any other system that is able to reproduce these results. We also derived consistent data decompositions for all 6 permutations of Gaussian elimination, and for various loop restructurings of adi. In fact, our system is *guaranteed* to produce the same results if we are able to correctly calculate the transitive dependences. If we use dependence relations to do so, we always can. If we use extended direction vectors, our calculations may not be exact, although we have observed this only in the case of imperfectly nested loops: for some permutations of Cholesky decomposition, extended direction vectors were insufficient. It should also be noted that our techniques depend on being able to accurately analyze the dependences in the program. If we cannot, perhaps due to the use of an indirection array, extensions beyond what are described in this paper will be necessary.

7 Related Work

Table 4 shows which of the desirable properties enumerated in Section 1 hold for a number of related works. Our work is most distinguished from all other related work by the fact that we are not influenced by the order of the computation in the original program. Most related works estimate parallelism and/or partition the program into phases based on the original loop structure.

Feautrier's approach [Fea94] is to find a schedule for executing the program with maximum parallelism, ignoring locality and latency. Then, he uses a greedy algorithm, based on the dimensionality of value-based dependences, to select a computation distribution that minimizes the volume of communications but doesn't place on the same virtual processor any two computations that could be run in parallel. The problem with this approach is that it is not possible to sacrifice some parallelism in a particular loop in order to reduce overall communication costs. By making use of pipelining, we can often obtain parallelism close to that afforded by a *doall* loop we decide to ignore.

Although other systems such as [RKU93, GAL95] also use exact rather than greedy heuristic algorithms, the size of the problems and the methods used are very different. We consider a list of candidate distributions for each statement, whereas these systems consider a list of candidate distributions for each array in each phase. Our search spaces will therefore tend to be much larger. These systems use 0-1 integer programming formulations, whereas we have developed our own graph search algorithm. The performance numbers given in [GAL95] (which uses a commercial 0-1 integer programming system called LINGO) tend to suggest that our search algorithm is significantly faster.

In Kremer's system [RKU93], an admittedly arbitrary

```
  for i = 1 to n
    for j = 1 to i-1
1     a(i,j) = a(i,j)/a(j,j)
      for k = 1 to j
2       a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
3   a(i,i) = sqrt(a(i,i))
```

$1 : \{[i,j] \rightarrow [i]\}$
$2 : \{[i,j,k] \rightarrow [i]\}$
$3 : \{[i] \rightarrow [i]\}$

```
  for i = 1 to n
    for k = 1 to i-1
1     a(i,k) = a(i,k)/a(k,k)
      for j = k+1 to i
2       a(i,j) = a(i,j)-a(i,k)*a(j,k)
3   a(i,i) = sqrt(a(i,i))
```

$1 : \{[i,k] \rightarrow [i]\}$
$2 : \{[i,k,j] \rightarrow [i]\}$
$3 : \{[i] \rightarrow [i]\}$

```
  for j = 1 to n
1   a(j,j) = sqrt(a(j,j))
    for i = j+1 to n
2     a(i,j) = a(i,j)/a(j,j)
      for k = 1 to j
3       a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
```

$1 : \{[j] \rightarrow [j]\}$
$2 : \{[j,i] \rightarrow [i]\}$
$3 : \{[j,i,k] \rightarrow [i]\}$

```
  for j = 1 to n
1   a(j,j) = sqrt(a(j,j))
    for i = j+1 to n
2     a(i,j) = a(i,j)/a(j,j)
    for k = 1 to j
      for i = j+1 to n
3       a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
```

$1 : \{[j] \rightarrow [j]\}$
$2 : \{[j,i] \rightarrow [i]\}$
$3 : \{[j,k,i] \rightarrow [i]\}$

```
  for k = 1 to n
1   a(k,k) = sqrt(a(k,k))
    for i = k+1 to n
2     a(i,k) = a(i,k) / a(k,k)
      for j = k+1 to i
3       a(i,j) = a(i,j) - a(i,k)*a(j,k)
```

$1 : \{[k] \rightarrow [k]\}$
$2 : \{[k,i] \rightarrow [i]\}$
$3 : \{[k,i,j] \rightarrow [i]\}$

```
  for k = 1 to n
1   a(k,k) = sqrt(a(k,k))
    for i = k+1 to n
2     a(i,k) = a(i,k)/a(k,k)
    for j = k to n
      for i = j+1 to n
3       a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
```

$1 : \{[k] \rightarrow [k]\}$
$2 : \{[k,i] \rightarrow [i]\}$
$3 : \{[k,j,i] \rightarrow [i]\}$

Figure 9: All six legal loop permutations for cholesky decomposition with selected space mappings

scheme is used to identify a sequential loop nest that contains a series of phases, which are executed atomically (i.e., without overlap). Parallelism is exploited within each phase but not between them. Using techniques not described in [RKU93], a set of candidate distributions are generated for each phase, and the system determines the cost of executing each phase in each distribution and the cost of the remapping variables between each transition. The system is very depended on obtaining a good partitioning of the program into phases and on having a good method to generate and evaluate distributions for each phase.

The system described in [GAL95] uses a static data decomposition for the entire program. They minimize communication volume and insure that the program can be executed in parallel simply by making one of the loops in the original program a *doall* loop. They do not consider transformations such as loop distribution or interchange and do not consider pipelined parallelism nor the differences in synchronization costs between different candidate loops.

8 Conclusion

We believe that we have succeeded in our goal of building a system that simultaneously optimizes for communication and parallelism without resorting to greedy or heuristic algorithms, and without being influenced by the order of the computation in the original program. Our system remains heuristic in one major way: we combine the effects of parallelism and communication simply by multiplying one by a constant parameter and then adding them together. This method of combination will be inaccurate if communication can be substantially overlapped with computation or with other communication. This heuristic was forced on us by a "chicken and egg problem": it is difficult to distribute the computations until the final order of the computations is known, but it is also difficult to order the computations until the distribution is known. Our heuristic works well in practice because the largest communications are unlikely to be substantially overlapped with computation. In future work, we will look into deriving multi-dimensional space mappings and will continue our work on determining the best order in which to execute iterations, given the space mappings determined here.

References

[AAL95] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[AL93] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, September 1994.

[GAL95] Jordi Garcia, Eduard Ayguade, and Jesus Labarta. A novel approach towards automatic data distribution. In *Workshop on Automatic Data Layout and Performance Predition*, April 1995.

[Gup92] M. Gupta. *Automatic Data Partioning on Distributed Memory Multicomputers.* PhD thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1992.

[HKT91] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for FORTRAN D on MIMD distributed memory machines. In *Supercomputing '91*, November 1991.

[KP95] Wayne Kelly and William Pugh. Identifying reordering transformations that minimize idle processor time. Technical Report CS-TR-3431, Dept. of Computer Science, University of Maryland, College Park, February 1995.

[KPRS95] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. In *Eighth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[Pug94] William Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[PW93] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

[RKU93] R.Bixby, K.Kennedy, and U.Kremer. Automatic data layout using 0-1 integer programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, November 1993.

[SSP+95] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. In *Eighth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[WB87] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *Internation J. Parallel Programming*, 16(2):137–178, April 1987.

[Wol91] Michael Wolfe. Experiences with data dependence abstractions. In *Proc. of the 1991 International Conference on Supercomputing*, pages 321–329, June 1991.