A Methodology for Parallelizing Programs for Multicomputers and Complex Memory Multiprocessors

J. Ramanujam and **P. Sadayappan**
Department of Computer and Information Science
The Ohio State University, Columbus Ohio 43210

Abstract

The availability of large distributed memory multiprocessors and parallel computers with complex memory hierarchies have left the programmer with the difficult task of planning the detailed parallel execution of a program; for example, in the case of distributed memory machines, the programmer is forced to manually distribute code and data in addition to managing communication among tasks explicitly. Current work in compiler support for this task has focused on automating task partitioning, assuming a fixed data partition or a programmer-specified (in the form of annotations) data partition. This paper argues that one of the reasons for inefficient parallel execution is the lack of synergism between task partitioning and data partitioning and allocation; hence data and task allocation should both be influenced by the inherent dependence structure of the computation (which is the source of synergism). We present a methodology based on a **unified** approach to task and data partitioning; we show how to derive data and task partitions for computations expressed as nested loops that exhibit regular dependences and how to map these onto distributed memory multiprocessors. Based on the mapping, we show how to derive code for the nodes of a distributed memory machine with appropriate message transmission constructs. We also discuss related communication optimizations.

1 Introduction

Rapid developments in large scale parallel systems – in the form of shared memory multiprocessors with complex memory hierarchies such as IBM RP3, BBN Butterfly, Cedar etc., and distributed memory machines such as the hypercube machines from Intel, NCUBE, Ametek and FPS – have left the programmer or the algorithm designer with the difficult task of detailed planning of the computation, *i.e.* the **orchestration** of the entire parallel execution. In the context of distributed memory machines such as hypercubes, the programmer is forced to manually distribute code and data in addition to managing communication among tasks explicitly. This task, in addition to being error-prone and time-consuming, generally leads to non-portable code.

The enormity of the task is to some extent relieved by the *hypercube programmer's paradigm* where attention is paid to the partitioning of tasks alone while assuming a fixed data partition or a programmer-specified (in the form of annotations) data partition [10, 15]. Though these rely on the intuition (based on domain knowledge) of the programmer, it is not always possible to verify that the annotations indeed result in efficient execution. This paper argues that one of reasons for inefficient parallel execution is the lack of synergism (or "match") between task and data partitioning; hence data and task allocation should both be influenced by the inherent dependence structure of the computation (which is the source of synergism).

We propose a methodology to deal with the partitioning of data and tasks in a unified manner, based on dependence analysis of the program. The partitioning of tasks should be such that the computation is load-balanced over time, and the partitioning of data should result in reduction of the total communication required. These two criteria should be considered together in determining appropriate partitioning of tasks and data, rather than as independent issues as has often been done in the past. We show through an example that a user-defined partition of data that is optimal with respect to total communication and with respect to overall load-balancing may be non-optimal in terms of completion time on a distributed-memory system due to the isolated consideration of these two issues. We consider the class of programs expressible as nested loops with a regular dependence structure, and develop an approach to the derivation of data and task partitions for such computations. Based on the mapping, we show how to derive code for the processors of a distributed memory machine with appropriate message transmission constructs. The work presented here differs significantly from previous work in that we do not rely on a user-defined data partition, but attempt instead to derive a data partition based on an analysis of the dependence structure of the program. The proposed approach is applicable in the context of data parallel algorithms, which are abundant in scientific computation, signal and image processing.

The architectural model we consider is that of a distributed memory machine (there is no globally shared memory) with a crossbar interconnection topology (assumed for simplicity of exposition). The approach to process synchronization and communication that we assume is the *message-passing* model where processes communicate through *explicit* message passing in the form of *sends* and *receives.* As soon as a node computes a new value, it sends it to all those nodes which need the value. Thus the computation is data- or communication-driven; hence this style of programming has also been referred to as **large grain-size dataflow** [9].

The paper is organized as follows. In section 2, we discuss dependence analysis of nested loop programs and the hyperplane technique for identifying parallel threads of computation. Section 3 develops the approach to unified task and data partitioning. Section 4 illustrates the use of the technique using an example of a simple 2-dimensional relaxation code. Section 5 places this work in perspective, comparing it with other related work. We conclude in section 6 with a discussion of the scheme, and outline a systematic procedure for deriving "hypercube" algorithms based on systolic array synthesis techniques [8, 20].

2 Data Dependence and Parallelism

The automatic detection of parallelism involves finding sets of computations that can be performed simultaneously. The potential parallelism in a computation is determined by data dependences [17, 21, 22]. The presence of a dependence between two computations (statements) implies that they cannot be executed in parallel. These dependences imply precedence constraints among computations which have to be satisfied for a correct execution. Many algorithms exhibit regular data dependences, *i.e.* certain dependence patterns occur repeatedly over the duration of the computation.

2.1 Data Dependences

Consider a loop structure of the form:

for $I_1 = l_1$ to u_1 do
  for $I_2 = l_2$ to u_2 do
    ...
      for $I_n = l_n$ to u_n do
        $S_1(\vec{I})$
        $S_2(\vec{I})$
        ...
        $S_N(\vec{I})$

where l_j and u_j are integer-valued linear expressions involving $I_1, \ldots, I_{j-1}$ and $\vec{I} = (I_1, I_2, \ldots, I_n)$. $S_1, S_2, \ldots, S_N$ are assignment statements of the form $X_o = E(X_1, X_2, \ldots, X_k)$ where X_o is an output variable produced by expression E evaluated using some input variables $X_1, X_2, \ldots, X_k$. The input variables of a statement are produced by other statements, and sometimes by the same statement at an earlier execution. To each computation, we associate an integer vector index $\vec{L} = (I_1, \ldots, I_n)^T$. Thus, we have an index set:

$$L^n(\vec{I}) = \{(I_1, \ldots, I_n) : l_1 \leq I_1 \leq u_1, \cdots, l_n \leq I_n \leq u_n\}$$

When the loop is executed sequentially, the elements of L^n are in lexicographical ordering. This induced (sequential) ordering is non-essential and exposing and exploiting parallelism is about modifying this ordering (to reduce parallel execution time) while while preserving the semantics of the loop. Let f and g be two integer functions defined on the set L^n such that $X(f(\vec{I}))$ is a "defined" (output) variable and $Y(g(\vec{I}))$ is a "used" (input) variable. Consider two statements $S(\vec{I_1})$ and $S(\vec{I_2})$.

Flow Dependence: Statement $S(\vec{I_2})$ is flow dependent on $S(\vec{I_1})$ iff for $f(\vec{I_1}) = g(\vec{I_2})$ (where $\vec{I_1} \leq \vec{I_2}$), $X(f(\vec{I_1}))$ is an output variable in statement $S(\vec{I_1})$ and $X(g(\vec{I_2}))$ is an input variable in statement $S(\vec{I_2})$.

Anti-Dependence: Statement $S(\vec{I_2})$ is anti dependent on $S(\vec{I_1})$ iff for $g(\vec{I_1}) = f(\vec{I_2})$ (where $\vec{I_1} \leq \vec{I_2}$), $X(g(\vec{I_1}))$ is an input variable in statement $S(\vec{I_1})$ and $X(f(\vec{I_2}))$ is an output variable in statement $S(\vec{I_2})$.

Output Dependence: Statement $S(\vec{I_2})$ is output dependent on $S(\vec{I_1})$ iff for $f_1(\vec{I_1}) = f_2(\vec{I_2})$ (where $\vec{I_1} \leq \vec{I_2}$), $X(f_1(\vec{I_1}))$ is an output variable in statement $S(\vec{I_1})$ and $X(f_2(\vec{I_2}))$ is an output variable in statement $S(\vec{I_2})$.

The vector $\vec{d} = \vec{I_2} - \vec{I_1}$ is called the **iteration space data dependence vector** or simply **dependence vector.** Dependence relations are often represented in *Iteration Space Dependence Graphs* (ISDG's); for an n-nested loop with index set $I_1, I_2, \ldots, I_n$), the nodes of the ISDG are points on on an n-dimensional discrete Cartesian space and a directed edge exists between the iteration defined by $\vec{I_1}$ and the iteration defined by $\vec{I_2}$ whenever a dependence exists between statements in the loop constituting the iterations $\vec{I_1}$ and $\vec{I_2}$.

An algorithm may have a number of such dependence vectors. In general, the dependence vectors are functions of the index point $\vec{I}$. There is however, large class of algorithms with fixed, or constant or uniform data dependence vectors. The dependence vectors of an algorithm can be written collectively as a **dependence matrix** $D = [\vec{d_1}, \vec{d_2}, \ldots, \vec{d_m}]$. In addition to the three types of dependence above, there is one more type of dependence known as **control dependence**. A control dependence exists between a statement with a conditional jump and another statement if the conditional jump statement controls the execution of the other statement.

Of these data dependences, only flow dependence is inherent to the computation whereas the other two forms of dependence arise due to the storage-based imperative programming model. In our analysis, we consider distributed memory machines where considering flow dependences alone is sufficient. These data dependences constrain the order of execution of the statements in a program. Methods to calculate data dependence vectors can be found in [1, 2, 27, 29].

In this section, we consider program loops which can exhibit uniform or regular dependence structure – every index point has the same dependences – and to specify the dependences exhibited by the algorithm, you only need to specify the dependence of one index point. The dependence distances are constant (not functions of indices). For such computations, a variety of systolic array synthesis techniques exist, most of which analyze the dependence structure through linear transformations (affine if the distances are linear function of the indices) [8].

2.2 Hyperplane of Parallelism

In this section, we reiterate a fundamental result due to Karp et. al., [13] (dealing with parallel execution of computations expressed as uniform recurrence equations) and Lamport [18] (dealing with parallel execution of nested loops in FORTRAN) – this result has been used by Moldovan [8] , Miranker [20] and others. Lamport considered partitioning of the indices $\vec{I} = (I_1, I_2, \ldots, I_n)$ such that they lie on a family of parallel hyperplanes such that all indices lying on one hyperplane can be executed simultaneously. Let $\vec{H}$ be a vector $(t_1, t_2, \ldots, t_n)$. Then,

$$t_1x_1 + t_2x_2 + \cdots + t_nx_n = k$$

for various values of k defines a family of hyperplanes. We use the terms hyperplane, family of hyperplanes and the vector defining them interchangeably where the context is clear. A case of Lamport's result is the following:

Theorem 1 *Let $\vec{H} = (t_1, t_2, \ldots, t_n)$ be a hyperplane such that $H\vec{d_i} > 0$ for each dependence vector $\vec{d_i} = (d_{i1}, d_{i2}, \ldots, d_{in})^T$. If two indices $\vec{I_1}$ and $\vec{I_2}$ satisfy $HI_1 = HI_2 = c$, then $\vec{I_1}$ and $\vec{I_2}$ are independent of each other and hence can be executed simultaneously.*

We note that if no such H exists, then the computation defined by the nested loop involves deadlocks. In the event that communication costs are negligible, an optimal hyperplane $H^* = (t_1^*, t_2^*, \ldots, t_n^*)$ minimizes the number of hyperplanes required to partition the indices and gives the optimal time for a parallel implementation of the algorithm on an MIMD machines. Thus the execution time of the parallel algorithm is given by:

$$\frac{\max \quad H(\vec{I_1} - \vec{I_2}) + 1}{\min \quad H\vec{d_i}}$$

for any $\vec{I_1}, \vec{I_2} \in L^n$ and $d_i \in D$. Thus, the normal to the hyperplanes defines the direction of sequentiality; we refer to the (optimal) hyperplane henceforth as H and the normal to this hyperplane as the **progress vector.** All index points in a plane normal to the progress vector are independent. In the case of 2-dimensional index spaces, the hyperplane is also referred to as the **wavefront.** *Systolic arrays* are homogeneous networks of processors with local and regular interconnection and extremely fast inter-processor communication; each processor operates on data that flow regularly through the network. Thus a technique for identifying optimal hyperplanes (read efficient schedules) may thus be limited to *simple* schedules that exploit this structure; by a *schedule,* we mean the specification of time instances and processor assignments for the execution of individual statements. We note that good schedules can be derived using techniques described in [6, 7]. This scheme is more compact than performing a breadth first search of the iteration space dependence graph and assigning levels to node in such a way that all nodes at a given level can be executed in parallel.

In the context of systolic arrays, communication is extremely regular and fast leaving only the exploitation of parallelism to the designer; where as in the case of multiprocessors, communication (data access) is an important issue – in the case of shared memory machines with fast local memory, access to data stored in local memory is at least an order of magnitude larger and in the case of distributed memory message-passing machines such as Intel iPSC/2, message start-up times are much larger; the message start-up time for Intel iPSC/2 in 250 μs for a packet where as the transmission cost per pocket is about 10 μs, while access to local memory takes negligible time. Thus, exploiting *spatial* locality of reference is important and the trade-off between degree of parallelism and communication (data access) cost decides what a "good" schedule is. Thus, techniques developed for systolic arrays are not directly applicable in the context of multiprocessors.

In the next section, we outline the scheme for data and task partitioning in the context of mapping nested loop computations on distributed memory multiprocessors; the partitioning scheme is motivated by the preceding discussion.

3 A Data and Task Partitioning Scheme

There are two aspects that have to be paid attention to for efficient parallel execution of programs, namely,

- keeping the processors busy during the entire execution so that the load is as nearly uniformly distributed among processors over time.
- minimizing communication costs arising from partitioning the computation among processors.

The parallel execution time depends on the above two aspects as well as other issues such as network congestion, message startup times, the statistical distribution of message, the interconnection topology of the architecture onto which the computation is to be mapped etc.

3.1 Measures of quality of partitions

We develop a scheme that simultaneously addresses both task and data partitioning; we use a distributed memory multiprocessor with crossbar interconnection for the purpose of the discussion. Work is currently in progress in arriving at a more accurate model of the problem. At present, we use rather simplistic measures for determining the quality of a partition (mapping):

1. the degree of load balance over time, which is indicated by the parallel execution time if we ignore communication costs altogether
2. the communication cost, which is the total amount of communication incurred under the mapping.

3.2 Notations and Assumptions

We consider nested loops of the form:

for $I_1 = l_1$ to u_1 do
    $\cdots$

for $I_n = l_n$ to u_n do

$S_1(\vec{I})$

...

$S_N(\vec{I})$

where l_j and u_j are integer-valued linear expressions involving $I_1, \ldots, I_{j-1}$ and $\vec{I} = (I_1, I_2, \ldots, I_n)$. $S_1, S_2, \ldots, S_N$ are assignment statements of the form $X_o = E(X_1, X_2, \ldots, X_k)$ where X_o is an output variable produced by expression E evaluated using some input variables $X_1, X_2, \ldots, X_k$. When a variable appears on the left-hand side of an assignment statement in the loop body, it is said to be *defined* and when it appears in the right-hand side of an assignment statement, it is said to be *used.* We make the following assumptions about the loop body:

1. The expression E is not a function call and does not involve function calls.
2. Each occurrence of a defined or used array variable must contain only linear (affine to be more precise) functions of the loop indices and each subscript of the array variable is an affine function of at most one loop index. Thus all array references (for array A) are of the form:

$$A[f_1(I_1, \ldots, I_n), f_2(I_1, \ldots, I_n), \ldots, f_n(I_1, \ldots, I_n)]$$

   where each function f_j is of the form:

$$f_j(I_1, \ldots, I_n) = \alpha_j + \beta_j \times I_k \quad \text{where} \quad 1 \leq k \leq n$$

   While this assumption may seem restrictive, this represents the most common occurrences of subscripts in scientific computation.
3. Since we only use dependences and assume a single-assignment execution, in the case of n-dimensional arrays referenced in n-nested loops, no two subscripts should be functions of the same loop index.

3.3 Subscript Analysis and Data Space Dependence Graph

The subscript analysis of array references and relation between various points in the iteration space graph have a certain relation that is captured as:

> For loops with constant and regular dependence distance vectors, array references that satisfy the assumptions listed in the previous section form Data Space Distance Vectors (DSDV) that are described by families of planes defined by:
>
> $$a_{j,1}I_1 + a_{j,2}I_2 + \cdots + a_{j,n}I_n = c_{j,k}$$
>
> where $1 \leq j \leq m$ (m is the number of dependence vectors and n is the dimension of the index set) for values $c_{j,k}$.

In general, one must construct an Data Space Dependence Graph (DSDG) defined over the array subscript index set. But, for loops with regular and constant (iteration) dependence distance vectors, this is succinctly described by the Data Space Distance Vectors (DSDV); we will refer to them as **chain-vectors.** (Note that **chain vectors** may span different arrays, *i.e.* if $B[i]$ is needed in computing $A[i, j]$ then there is a chain vector from $B[i]$ to $A[i, j]$ for all i and j).

3.4 The Partitioning Scheme

Our partitioning scheme is motivated by the following observations:

1. In the **Iteration Space Dependence Graph,** the progress vector indicates the critical path in the computation; hence intuitively, the partitioning into tasks should "align" itself with the progress vector and index points (referring to statements) along the hyperplane H should be partitioned among the processors equally.
2. The communication cost incurred under the partition (mapping) depends on the number of data transfers that are "internalized" in the **Data Space Dependence Graph** as a result of the partition – *i.e.* the number of intra-partition communications; the larger this number, the lower the communication cost. The data transfers are indicated by the chain vectors. This suggests that we should "align" the data partitions with the chain vectors; the choices here are: one could align the data partitions along one of the chain vectors so that data transfer costs due to that vector are eliminated. The second choice we have is to align the data partitions along the "composite" chain vector (which is the sum of the individual chain vectors) so that "internalization" of data transfers is shared among the different chain vectors. A third choice is to partition the data (array) into blocks or strips oriented (by a transformation of indices) along a certain direction that minimizes the number of chain-vectors that are incident on two partitions.

The proposed scheme works in general as follows:

1. Perform a data dependence analysis of the nested loop program considering only flow dependences.
2. Find the **progress vector**.
3. Identify the index sets along each hyperplane of parallelism.
4. Evaluate the following partitions with respect to the two measures keeping in mind that among those partitions that balance load among processors in time, the desirable one is which minimizes communication; the partitions to be evaluated are derived from :
   - data partition from projection along the composite chain vector
   - data partition from projections along each of the individual chain vectors
   - data partition into blocks or strips oriented along a chosen direction that results in minimum number of inter-partition edges; for some cases of dependences, this is a *block-* version of case 1.

- partitions from projections along the array axes, *i.e.* in terms of k dimensional subspaces of an n-dimensional array for values of k from 1 to n.

5. The partitions are mapped onto nodes of the target machine.
6. Identify the set of non-local references for each processor node; this is the set of all edges in the data-space graph, the vertices of which are incident on two partitions. Geometrically, this is is equivalent to constructing the local and non-local components of the *reference window.*
7. Perform communication optimization by an analysis of the communication-parallelism trade-off taking into account the derived partitions and the machine characteristics (parameters); for hierarchical shared memory machines, the parameters are the size of the local cache, access time for global vs. local access and and time to transfer data from global to local memory; for message-passing multicomputers, the parameters are message startup-time, per word transmission cost and network congestion. For example, in the latter case, a result of communication optimization would be to send a block of values, not one after the other (as soon as it is defined).
8. Given the partitions and the optimization from the last step, insert appropriate **sends** and **receives.** (This is discussed in the next section in the context of the example.)

3.5 Illustration of the scheme

Consider the following nested loop: (k is some known constant scalar)

Loop 1:
  for $i = 1$ to N
    for $j = 1$ to N
      $A[i,j] \leftarrow A[i-1,j] + k$

The iteration space graph for Loop 1 is shown in Figure 1.(a) and its data space graph in Figure 1.(b). The ISG dependence vector is $(1,0)^T$. The loop has N independent threads of computation. From the data space graph, it is obvious that a column-partition of matrix A is optimal with respect to communication and load-balance; a row or block partition increases communication cost dramatically.

Now, consider a minor variant of the same loop: (k is some constant scalar)

Loop 2:
  for $i = 1$ to N
    for $j = 1$ to N
      $A[j,i] \leftarrow A[j,i-1] + k$

The iteration space graph for Loop 2 is shown in Figure 2.(a) and its data space graph in Figure 2.(b). The ISG dependence vector for this loop is also $(1,0)^T$. ¿From the data space graph, it is obvious that a row-partition of matrix A is optimal with respect to communication and load-balance; a column or block partition increases communication cost dramatically. Recall our assumption that the dimension of the arrays is the same as the nesting level of the loop and that no two subscripts of the same array are functions of the same loop index variable. To this if we add the condition that all subscripts are of the form $I_k + c$ where c is an integer constant, we have a case where the Iteration Space Graph and the Data Space Graph can be described by the same set of vectors.

Consider the following fully parallel loop:

Loop 3:
  for $i = 1$ to N
    for $j = 1$ to N
      $A[3i,2j] \leftarrow A[3i-1,j] + A[i,j]$

The iteration space graph for Loop 3 is shown in Figure 3.(a) and its data space graph in Figure 3.(b). The ISG dependence vector is $(0,0)^T$. The loop is fully parallel; thus any partition of the data space, into rows, columns or blocks that does not cut edges in the data space graph and that results in load-balanced partition would suffice.

In the above loops, a transformation of the array subscripts was not called for. Consider the following fully parallel loop:

Loop 4:
  for $i = 1$ to N
    for $j = 1$ to N
      $A[3i,2j] \leftarrow A[3i-2,2j-1]$

The iteration space graph for Loop 4 is shown in Figure 4.(a) and its data space graph in Figure 4.(b). The ISG dependence vector is $(0,0)^T$. The loop is fully parallel; but one has to be very careful in partitioning the data space – the standard row, column or block partitions would entail unnecessary communications. In this case the partitions should be along the direction given by the chain-vector $(2,1)^T$ where the first and second components of the vector refer to i and j respectively. This array can be transformed to an array AA where the partitions are by rows, columns or blocks.

In the above loops, there was exactly one chain-vector; in the following loop, there are 3 chain-vectors resulting in a block partition of a transformed array (row and column partitions result in higher communication costs).

Loop 5:
  for $i = 1$ to N
    for $j = 1$ to N
      A[3i,2j] ← A[3i-3,2j-2] + A[3i-3, 2j] + A[3i,2j-2]

The iteration space graph for Loop 5 is shown in Figure 5.(a) and its data space graph in Figure 5.(b). The ISG dependence vectors are $(0,1)^T$ and $(1,0)^T$. There are 3 chain vectors $(0,2)^T$, $(3,0)^T$ and $(3,2)^T$. To minimize communication, we choose a block partition along the direction given by $(3,2)^T$.

4 Example Mappings: 2-D relaxation

Consider the following loop which is a simple case of a 2-dimensional relaxation code:

for $i = 1$ to n
  for $j = 1$ to n
    $u(i,j) \leftarrow u(i-1,j) + u(i,j-1)$

There are two dependences: from $u(i-1,j)$ to $u(i,j)$ and from $u(i,j-1)$ to $u(i,j)$. Thus the dependence vectors are $\vec{d_1} = (1,0)^T$ and $\vec{d_2} = (0,1)^T$ where the first component is indexed with i and the second with j to indicate the dimension along which the dependence exists.

The optimal hyperplane in this case is given by $i + j =$ constant for different value of the constant from 2 through $2n$. The vector H corresponding to this is given by $\vec{H} = (1,1)^T$. It is easy to verify that $\vec{H}\vec{d_i} > 0$ for $i = 1,2$. Thus, the progress vector is $\vec{H} = (1,1)^T$; in this case, we have a wavefront marching across a 2-dimensional grid of size $n \times n$ at an angle of 45 degrees. The dependence structure of the computation is shown in Figure. 6(a). The Data Space (array reference) dependence graph (DSDG) for this example is the same as the dependence graph and is shown in Figure. 6(b). (Hence the terms for the two graphs are used interchangeably.) The amount of parallelism available in the code is a function of the problem size n and the angle of incidence of the wavefront at the edges of the grid; the parallelism profile for this 2-D code is illustrated if Figure. 7. We now illustrate that minimizing communication without paying attention to load balance over time leads to inefficient parallel execution. Given such a 2-dimensional grid of points, an analysis of the communication requirements with a view to balance the load (but not through time), indicates that the matrix should be partitioned into p submatrices of size $n/\sqrt{p} \times n/\sqrt{p}$ for execution on p processors. With such a data decomposition for distributed memory machines, results in a task partition (which is the same as the data partition) as shown in Figure 8. With this partition,

$$\text{total communication cost} = 2n(\sqrt{p} - 1) \times T_d,$$

where T_d is the time taken for sending a data item and assuming zero communication cost,

$$\text{parallel computation cost} = T_e \left[\frac{3n^2}{2\sqrt{p}} - \frac{n^2}{2p} \right]$$

where T_e is the time taken to compute the value of a grid point. The reason for $n^2/\sqrt{p}$ term is that only $\sqrt{p}$ processors are active at the time of maximum parallelism (when the wavefront sweeps the grid at its widest) where as it is possible to derive other partitions where the computation cost is $O(n^2/p)$.

In this case, the progress vector and the composite chain vector are the same; in addition partitioning along the i index of the loop is symmetric to partitioning along the j index and partitioning along the chain vector $(0,1)^T$ is the same as partitioning along j and that partitioning along the chain vector $(1,0)^T$ is the same as partitioning along i.

In the ensuing discussion, for sake of simplicity we assume that processors are numbered 0 thru $p-1$ (PN stands for processor) and that n is a multiple of p. In the case of a hypercube topology we assume that both n and p are powers of 2. We impose the restriction that the processor to which a data item is assigned, *i.e.* the **MAP** is a function of the grid indices (i and j in this case), the size of the entire data (n here), the number of processors (p here) and the only operations allowed in the function are **mod, floor, ceiling** in addition to the standard arithmetic functions. This restriction is to help in inserting **sends** and **receives** which when executed by a processor do not require a table lookup; the complexity of the **MAP** may also be a consideration in comparing different partitions. Thus the partitions that are examined are:

Case 1: The partition induced by

$$MAP(u_{ij}) = PN_k \quad \text{where} \quad k = \left\lfloor \frac{j}{n/p} \right\rfloor$$

Here the grid is partitioned into strips. The computation cost is $T_e \times [2n^2/p]$ and the communication cost is $T_d \times 2n(p-1)$. The partition is shown in Figure. 9.

Case 2: The partition induced by

$$MAP(u_{ij}) = PN_k \quad \text{where} \quad k = \left\lfloor \frac{j}{b} \right\rfloor \bmod p$$

where b is referred to as the block size; $b$isaninteger ≥ 1 and $b = \frac{n}{xp}$ which means $1 \leq x \leq n/p$. The partition in case 1 is a special case of this for $b = \frac{n}{p}$. For the case of $b = \frac{n}{2p}$, the computation cost is $T_e \times \frac{n^2}{4p^2}\left[6p + \frac{3n}{2p} - 2\right]$ and the communication cost is $T_d \times n(2p-1)$. For the case that $b = 1$, we have the computation cost is $T_e \left[\frac{n^2}{p} + n\right]$ and the communication cost is $T_d \times 2n(p-1)\left[1 + \frac{n}{p}\right]$. Thus we see a trade-off in that as b increases from1 to $\frac{n}{p}$, the load balance improves, there by decreasing parallel computation time where as communication costs increase resulting in a factor of $\frac{n}{p}$. The partition for the case of $b = \frac{n}{2p}$ is shown in Figure. 10.

Case 3: Partitioning into square blocks; this corresponds to partitioning into smaller 2-dimensional subspaces. This case was discussed earlier.

Case 4: Partitioning data and tasks along the wavefront or the **progress vector** which happens to be the same as the **composite chain vector.** The restriction on the kind of operations allowed in the **MAP** function poses some problems here and we note that it is possible to derive slightly better mappings in terms of load-balance. The partition is given as:

$$MAP(u_{ij}) = PN_k \quad \text{where} \quad k = \left\lfloor \frac{j-i}{n/p} \right\rfloor \bmod p$$

Here is the grid is partitioned into strips aligned along its diagonal. The computation cost is $T_e[\frac{2n^2}{p} - \frac{n^2}{p^2}]$ and the communication cost is $T_d \times 2n(p-1)$. The partition is shown in Figure. 11.

Case 5: This is similar to case 4 except that the block size b is not n/p but any integer ≥ 1 and and $b = \frac{n}{xp}$ which means $1 \leq x \leq n/p$. In this case, the communication cost increases by a factor of x while the computation cost approaches $T_e \times n^2/p$.

Given these cases which follow from the scheme, the best one depends on the values of T_d and T_e and the interconnection topology.

Code Derivation for 2-D relaxation

Recall that the data- (communication-) driven programming model that we have assumed dictates that as soon as a processor computes a new value, it sends it to all those processors who need the value. Thus prior to each computation, the processor has to check if it is making a non-local reference; if so, it must, perform a **receive.** After, the computation, it must perform a **send** to those who need the value.

Consider the mapping of case 1, shown in Figure. 9; here

$$MAP(u_{ij}) = PN_k \quad \text{where} \quad k = \left\lfloor \frac{j}{n/p} \right\rfloor$$

Hence, **MAP** is independent of i and processor k executes the statement for i from 1 to n and from $j = k * (\frac{n}{p}) + 1$ to $(k+1)\frac{n}{p}$. Since for the execution of the statement for u_{ij}, it needs $u_{i,j-1}$ and $u_{i-1,j}$, it executes a **receive** statement. Let d_m stand for the m-th dependence vector and let the index set have s dimensions; let $d_{m,s}$ refer to the s-th component of the m-th dependence vector. In this case $m = 2$ since there are two dependence vectors $\vec{d_1} = (0,1)^T$ and $\vec{d_2} = (1,0)^T$ and the index set is of dimension 2. Since each processor executes the same code, the value $u(i,j)$ (we use $u(i,j)$ in stead of $u_{i,j}$ for the sake of clarity) is needed at $MAP(u(i+d_{1,1}, j+d_{1,2}))$ and $MAP(u(i+d_{2,1}, j+d_{2,2}))$, *i.e.* it is needed at $MAP(u(i+1,j))$ and $MAP(u(i,j+1))$. Hence, a **send** is performed if these values are non-local. Thus, the code for a non-boundary processor k $(1 \leq k \leq p-2)$ is:

for $i = 1$ **to** n
    for $j = (k \times \frac{n}{p} + 1)$ **to** $(k+1) \times \frac{n}{p}$
        if $MAP(u(i,j-1)) \neq k$ **then**
            receive $(u(i,j-1), MAP(u(i,j-1)))$
        if $MAP(u(i-1,j)) \neq k$ **then**
            receive $(u(i-1,j), MAP(u(i-1,j)))$
        $u(i,j) \leftarrow u(i-1,j) + u(i,j-1)$
        if $MAP(u(i,j+1)) \neq k$ **then**
            send $(u(i,j+1), MAP(u(i,j+1)))$
        if $MAP(u(i+1,j)) \neq k$ **then**
            send $(u(i+1,j), MAP(u(i+1,j)))$

Communication Optimization

The non-local and local references for the processor are determined from the partitions as follows: the chain vectors are uniform over the array, therefore, the non-local values to be received are at the left-most column for each processor and the non-local values to be sent are at the right-most column for each processor (except for the boundary processor). Thus, the code for processor k $(1 \leq k \leq p-2)$ can be rewritten as:

for $i = 1$ **to** n
    receive $(u(i, k \times \frac{n}{p}), MAP(u(i, k \times \frac{n}{p})))$
    for $j = (k \times \frac{n}{p} + 1)$ **to** $(k+1) \times \frac{n}{p}$
        $u(i,j) \leftarrow u(i-1,j) + u(i,j-1)$
    endfor
    send $(u(i,(k+1) \times \frac{n}{p}), MAP(u(i,(k+1) \times \frac{n}{p} + 1)))$
endfor

Depending on the relative values of n and p and the message start-up time vs. transmission time, the **send**'s and **receive**'s will be done for a block, bi of outer loop iterations $1 \leq bi \leq n$. We are investigating various heuristics for this problem.

The following loop contains a relaxation expression for four point relaxation:

for $i = 1$ to l
    for $j = 1$ to n
        for $k = 1$ to n
            u[j,k] $\leftarrow$ 0.25 (u[j+1,k]+u[j,k+1]+u[j-1,k]+u[j,k-1])

In the loop statement, the index i is omitted but it is implied that when computing $u[j,k]$, we use $u[j+1,k]$ and $u[j,k+1]$ from the previous iteration *i.e.* $u[i-1,j+1,k]$ and $u[i-1,j,k+1]$ respectively and $u[j-1,k]$ and $u[j,k-1]$ from the current iteration. In this case, the scheme examines the partitions for 5 cases similar to the example discussed above; since, for large values of l, *i.e.* values of l of $O(n)$, all the partitions achieve the same degree of load balance, the block partition which minimizes communication is chosen.

5 Related Work

Chen et al. [5] describes how Crystal, a functional language addresses the issue of programmability and performance of parallel supercomputers. Gallivan et al. [10] discuss problems associated with automatically restructuring data so that it can be moved to and from local memories in the case of shared memory machines with complex memory hierarchy. They present a series of theorems that enable one to describe the structure of disjoint sub-lattices accessed by different processors and how to use this information to make "correct" copies of data in local memories and how to write the data back to the shared address space when the modifications are complete. Gannon et al. [11] discuss program transformations for effective complex-memory management for a CEDAR-like architecture with three-level memory. In the context of hierarchical shared memory systems, Irigoin and Triolet [12] present a method which divides the iteration space into nodes with the following goals: vector computation within a node, data re-use within a partition and parallelism between partitions. Koelbel et al. [15, 16] address the problem of automatic process partitioning of programs written in a functional language called BLAZE given a user-specified data partition. A group led by Kennedy at Rice University [3] is studying similar techniques for compiling a version of FORTRAN 77 that includes annotations for specifying data decomposition, for the Intel iPSC/2; they show how some existing transformations could be used to improve performance. Rogers and Pingali [23] present a method which, given a sequential program and its data partition, performs task partitions to enhance spatial locality of reference. Mace [19] proves that the problem of finding optimal data storage patterns for parallel processing (the "shapes" problem) is NP-complete, even when we limit ourselves to one- and two-dimensional arrays; in addition, efficient algorithms are derived for the shapes problem for programs modeled by a directed acyclic graph (DAG) that is derived by series-parallel combinations of tree-like subgraphs. Wolfe [28] discusses a technique referred to as *iteration space tiling* which divides the iteration space of loop computations into *tiles* (or blocks) of some size and shape so that traversing between the tiles results in covering the whole space; optimal tiling for a memory hierarchy will find tiles such that all data for a given tile will fit into the highest level of memory hierarchy and exhibit high data reuse, reducing total memory traffic.

6 Discussion

In this paper we discussed issues relating to the enormous task faced by the programmer (who has been served poorly by the availability of multiprocessors with complex memory hierarchies) of a full **orchestration** of parallel execution of a program; previous efforts [11, 15] to help in this task has focused on automating task partitioning assuming a fixed data partition or a programmer-specified (in the form of annotations) data partition. We argue one of reasons for inefficient parallel execution is the lack of synergism between task and data partitioning and allocation; and that both data and task allocation should be influenced by the inherent dependence structure of the computation (which is the source of synergism). We show how to derive data and task partitions for computations expressed as nested loops that exhibit regular dependences and how to map these onto distributed memory multiprocessors.

In the case of distributed memory machines with a hypercube interconnection topology, given the partitions derived from the scheme, mapping them onto the nodes of the hypercube can be viewed as a graph matching problem [25]; we are investigating the use of Kernighan-Lin [14] like local search heuristic with "hill-climbing" ability for this problem: an initial random mapping is successively improved by using a sequence of changes rather than a single change in the mapping that reduces the communication cost [24].

This approach can also be extended for shared memory multiprocessors with hardware support for data-level synchronization such as the HEP multiprocessor [26]; in these cases, the cost of unit communication is proportional to $log_2 p$ for making copies to the local cache; work is in progress for dealing with complex memories as well in implementing and evaluating the scheme in a **parallelizing compiler** for distributed memory machines.

References

[1] R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Trans. Programming Languages and Systems,* Vol. 9, No. 4, 1987, pp. 491-542.

[2] U. Banerjee, *Dependence Analysis for Supercomputing,* Kluwer Academic Publishers, Boston, MA, 1988.

[3] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing,* Vol. 2, Oct. 1988, pp. 151-169.

[4] M. Chen, "A Design Methodology for Synthesizing Parallel Algorithms and Architectures," *Journal of Parallel and Distributed Computing,* Vol. 3, Dec. 1986, pp. 461-491.

[5] M. Chen, Y. Choo and J. Li, "Compiling Parallel Programs by Optimizing Performance," *The Journal of Supercomputing,* 2, 1988, pp. 171-207.

[6] R. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 84-1177, (UMI 85-02121) Oct. 1984.

[7] R. Cytron, "Limited Processor Scheduling of Doacross Loops," *Proc. 1987 International Conference on Parallel Processing,* S. Sahni (Ed.), Aug. 1987, pp. 226-234.

[8] J. Fortes and D. Moldovan, "Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms," *Journal of Parallel and Distributed Computing,* Vol. 2, 1985, pp. 277-301.

[9] G. Fox et al., "Solving Problems on Concurrent Processors," Volume 1: General Techniques and Regular Problems, Prentice-Hall, Englewood Cliffs, California, 1988.

[10] K. Gallivan, W. Jalby and D. Gannon, "On the Problem of Optimizing Data Transfers for Complex Memory Systems," *Proc. 1988 ACM International Conference on Supercomputing,* St. Malo, France, pp. 238-253.

[11] D. Gannon, W. Jalby and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations," *Journal of Parallel and Distributed Computing,* Vol. 5, No. 5, October 1988, pp. 587-616.

[12] F. Irigoin and R. Triolet, "Supernode Partitioning," in *Proc. 15th Annual ACM Symp. Principles of Programming Languages,* San Diego, CA, Jan. 1988, 319-329.

[13] R.M. Karp, R.E. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM,* Vol. 14, Jul. 1967, pp. 563-590.

[14] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical Journal,* Vol. 49, No. 2, 1970, pp. 291-308.

[15] C. Koelbel, P. Mehrotra and J. van Rosendale, "Semi-automatic Process Partitioning for Parallel Computation," *International Journal of Parallel Programming,* Vol. 16, No. 5, 1987, pp. 365-382.

[16] C. Koelbel and P. Mehrotra, "Semi-automatic Process Decomposition for Non-shared Memory Machines," in *Proc. 1989 International Conference on Supercomputing,* May 1989.

[17] D. Kuck, R. Kuhn, B. Leasure, D. Padua and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proc. ACM 8th Annual Symposium on Programming Languages,* Williamsburg, VA, Jan. 1981, pp. 207-218.

[18] L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM,* Vol. 17, No. 2, Feb. 1974, pp. 83-93.

[19] M. Mace, "Memory Storage Patterns in Parallel Processing," Kluwer Academic Publishers, Boston, MA, 1987.

[20] W. Miranker and A. Winkler, "Spacetime Representation of Computational Structures," *Computing,* 32, 1984, pp. 93-114.

[21] D. Padua, D. Kuck and D. Lawrie, "High-speed Multiprocessors and Compilation Techniques," *IEEE Trans. Computers,* Vol. C-29, No. 9, Sep. 1980, pp. 763-776.

[22] D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM,* Vol. 29, No. 12, Dec. 1986, pp. 1184-1201.

[23] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference," *Proc. ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* Portland, OR, Jun. 1989, pp. 69-80.

[24] P. Sadayappan, F. Ercal and J. Ramanujam, "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube," to appear in *Parallel Computing.*

[25] C. Shen and W. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minmax Criterion," *IEEE Trans. on Computers,* Vol. C-34, No. 3, Mar. 1985, pp. 197-203.

[26] B.J. Smith, "A Pipelined, Shared Resource MIMD Computer," *Proc. 1978 International Conference on Parallel Processing,* pp. 6-8.

[27] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* Pitman Publishing, London and MIT Press, 1989.

[28] M. Wolfe, "Iteration Space Tiling for Memory Hierarchies," in *Parallel Processing for Scientific Computing,* G. Rodrigue (Ed.) SIAM, Philadelphia PA, 1989, 357-361.

[29] M. Wolfe and U. Banerjee, "Data Dependence and Its Application to Parallel Processing," *International Journal of Parallel Programming,* Vol. 16, No. 2, 1987, pp. 137-178.