Compile-Time Partitioning of Iterative Parallel Loops to Reduce Cache Coherency Traffic

Santosh G. Abraham, *Member, IEEE*, and David E. Hudak

Abstract— **In order to sustain processor–memory bandwidth and latency requirements, multiprocessors will have increasingly complex memory hierarchies. Effective compile-time partitioning schemes are essential to reduce communication and synchronization costs and achieve acceptable performance. Adaptive data partitioning (ADP) is proposed to reduce the execution time of parallel programs by reducing interprocessor communication for iterative parallel loops. ADP can be integrated into a communication-reducing back end for existing parallelizing compilers or as part of a machine-specific partitioner for parallel programs. A multiprocessor model is defined to analyze program execution factors that lead to interprocessor communication. A model for the iterative parallel loop is defined to quantify communication patterns within a program. A vector notation is chosen to quantify communication across a global data set. Communication parameters are computed by examining the indexes of array accesses and are adjusted to reflect the underlying system architecture by compensating for cache line sizes. These values are then used to generate rectangular and hexagonal partitions that reduce interprocessor communication. Experiments measuring the impact of communication on program performance were conducted using the ADAPT system, a program partitioner which automatically restructures Fortran code. ADP is shown to be superior to existing partitioning strategies by measuring execution times on a 16-processor Encore Multimax. Simulation results on a multiprocessor simulator indicate that ADP outperformed established data partitioning methods by reducing cache invalidate traffic thereby decreasing miss ratios by up to 30%.**

Index Terms— **Cache coherency, data placement, interprocessor communication, optimizing compilers, parallel loops, private-cache multiprocessors, program partitioning.**

I. Introduction

At the onset of parallel processing, research in compiling for parallel machines focused on the identification and utilization of parallelism. Experience on real multiprocessor systems uncovered new problems that have to be addressed in order to facilitate wider use of multiprocessor systems. In particular, interprocessor communication and synchronization can greatly degrade the achieved performance levels even when adequate parallelism is available and exploited.

Shared-memory multiprocessors offer a familiar model for programmers. However, compromises are necessary in building a scalable multiprocessor. Multiprocessor systems require complex hierarchies to meet processor–memory bandwidth and latency requirements. Parallel programs must be partitioned to exploit the memory hierarchy configuration of the target system. Future work in compiling for multiprocessor systems will involve automatic restructuring for the minimization of communication and synchronization.

In *parallel program partitioning*, a program is split into *tasks* that can execute concurrently on a multiprocessor. Static program partitioning is attractive since the program partition is specified during the compilation phase, thereby eliminating one source of run-time overhead. Furthermore, static partitionings reduce communication overhead relative to dynamic schemes that attempt to improve processor utilization. *Adaptive data partitioning* (ADP) is a static partitioning approach, which optimally divides a large global data set among concurrently executing processors to reduce communication requirements for particular parallel programming constructs.

Parallelizing compilers have been based on the following two premises. First, exploitable parallelism that scales with the problem size is largely present in DO loops, and therefore the development of suitable parallelization techniques for DO loops is of primary importance. Second, though parallelization techniques are based on the theoretical framework of dependence analysis, a range of techniques for commonly occurring programming constructs have to be developed. Our work on reducing interprocessor communication is also based on these two assumptions. Furthermore, since DO loops are most amenable to analysis, initial efforts at minimizing communication should be based on DO loops.

ADP will be useful in the development of portable parallel programs. Typically, users run their applications on a range of multiprocessors. Often, they are forced to modify their programs extensively to exploit specific memory hierarchies. An ideal compiler for portable parallel programs should remove this burden from the user. ADP, which partitions a programming construct automatically for a given multiprocessor, is a step toward such a compiler.

Consider Fig. 1, which is an instance of an *iterative parallel loop*. It consists of outer sequential DO loops and inner parallel loops performing updates on a large data set, an array. The outer sequential DO loop enforces synchronization at the end of each outer iteration and forces the sequentialization of updates to a particular element of the array. The iterations of the inner parallel loops may be executed in any order. An example of such a loop in numerical programs is the solution of partial differential equations through relaxation. The collection of array offsets is referred to as the *stencil*. Data partitioning in iterative parallel loops divides the index sets of the inner loops among processors. Each processor executes the same code on

Manuscript received October 2, 1990; revised February 24, 1991. This work was supported by the National Science Foundation under Grant ASC-8808980.

The authors are with the Department of Electrical Engineering and Computer Science, The University of Michigan Ann Arbor, MI 48109.

IEEE Log Number 9100388.

1045–9219/91/0700–0318$01.00 © 1991 IEEE

```
DOSEQ k = 1, 1000
  DOALL j = 1, n
    DOALL i = 1, n
      A(i,j) = (A(i+2,j) + A(i+1,j) + A(i-1,j) + A(i-2,j) + A(i,j+1) + A(i,j-1))/6.0
    ENDDOALL
  ENDDOALL
ENDDOSEQ
```

Fig. 1. Example of an iterative parallel loop.

a contained area of the data set, typically communicating only with processors that work on neighboring areas.

In our partitioning procedure, the stencil set (obtained from the program) and the cache line size (an architectural parameter) are used to compute communication parameters. Rectangular or hexagonal partitions are obtained from the values of the communication parameters. In contrast to previous work, rectangles and irregular hexagons are shown to be superior to squares and regular hexagons if either the communication is unequal in different directions or if the cache line size is a multiple of the size of each data point. In the presence of unequal communication and cache line effects, our algorithm may combine the two candidates to yield an optimal partition that is a square. We show that communication can be quantified by examining the array subscripts.

Previous data partitioning work and its relation to ADP is presented. Then, assumptions about the system and the program are discussed. A method of analyzing iterative parallel loops to construct communication patterns is described. In order to allow for comparisons between partitions, a communication metric is defined. Given the communication pattern of a code segment, formulas are presented to generate the partition dimensions that reduce this metric. A program partitioning system, ADAPT (Automatic Data Allocation and Partitioning Tool), is used to implement ADP. An analysis of a sample problem is presented, including simulation results and code timings from an Encore Multimax.

II. Related Work

We have drawn on three major sources of previous work: compiler techniques for partitioning parallel loops into independent execution sets, static methods for partitioning iteration spaces, and partitioning techniques for parallel algorithms. The first set of papers examines loops to determine sets of iterations that can be executed without any exchange of information between sets. The iteration space partitioning work simultaneously considers parallelization and partitioning. The algorithmic work partitions a global matrix by tiling it with one of a number of standard geometric shapes, e.g., squares, regular hexagons, or triangles.

Recent compiler-oriented work has been reported in [1]–[4]. The array offsets in the statements enclosed by a set of loops are examined. The index set is partitioned into independent execution sets, each of which can run on a distinct processor without interprocessor communication. In contrast to the previous work, which yields a variable number of independent execution sets, ADP partitions a parallel loop to achieve utilization of all the available processors, incurring small (but not zero) amounts of interprocessor communication. ADP is complementary to independent execution set partitioning. When an insufficient number of independent execution sets are available, ADP, applied on individual independent execution sets, increases processor utilization at the cost of some communication.

A good deal of work has been done in the tiling of sequential loop iteration spaces [5]–[9]. A common thread in all of these papers is the simultaneous consideration of parallelization and partitioning using dependence-oriented approaches. Unlike these previous methods, adaptive data partitioning is an optimization for a parallel construct. The partitioning obtained using ADP is optimal under a metric. Previous work does not attempt to measure the quality of a partitioning or prove its optimality. Detection of parallelism is a "front-end" activity, i.e., possible without considering the specifics of the target multiprocessor. Conversely, effective partitioning is a "back-end" activity, requiring extensive knowledge of the machine. Therefore, we sought to separate the two activities.

Bus traffic in multiprocessor systems consists of two components: uniprocessor misses and cache coherency traffic [10]. Both Gallivan *et al.* [11], [12], and our work share a common goal of maximizing hit ratios through program transformations. They define a *reference window* for a dependence as the variables referenced by both the source and sink of the dependence. After executing the source of the dependence, saving the associated reference window in the cache until after the sink has been executed improves the number of hits. These optimizations attempt to maximize reuse of cache data in the context of a finite cache, i.e., reducing uniprocessor misses. ADP does not consider cache size effects and minimizes the number of variables shared among processors, thus reducing cache coherency traffic.

Relevant work on partitioning at the algorithm level is discussed in [13]–[16], and [11]. Reed *et al.* [16] presents a thorough treatment of the topic, including a formal method for analyzing stencil/partition/architecture triplets in order to unify the effects of several factors on an algorithm's performance within one model. Other significant work includes that of Gallivan *et al.* [12], who attempt to balance effects of locality, concurrency, and vectorization on the performance of dense linear algebra algorithms, forming *block* algorithms which improve performance through increased locality.

Previous work on partitioning algorithms provides excellent guidelines for parallel algorithm design and parallel program development. ADP is a translation of these algorithmic techniques into a compiler technique that automatically generates optimal program partitions. By removing the burden of memory management from the programmer, we hope to both shorten parallel program development time and improve parallel program performance.

This paper develops algorithms suitable for compiler use that detect communication patterns in code segments and adjust the shape of partitions to reduce communication for these segments. The work presented here takes ideas from previous approaches to data partitioning. We use geometric shapes to tile a global matrix, similar to algorithm partitioning. However, we alter the partition shapes to compensate for nonuniformities in the communication patterns, a factor not considered previously. It will be shown that some results presented in [16] and [14] are special cases of ADP.

III. Model Assumptions

A. Multiprocessor Model

The partitioning strategy presented here is effective for a range of parallel systems, including shared-memory and distributed-memory systems. The notable exceptions are shared-memory multiprocessors without caches or with shared caches, e.g., the Alliant FX-8. In order to quantify and illustrate the improvements in interprocessor communication attained by using adaptive data partitioning, we focus on shared-memory multiprocessors with private copy-back caches. Such multiprocessors are perhaps the most successful commercially, e.g., Sequent Symmetry and Encore Multimax.

Bus-based multiprocessor systems increasingly use a copy-back cache protocol in which each cache monitors, or "snoops" on the system bus to ensure memory coherency. The copy-back snooping protocols reduce traffic on the bus since repeated updates of a possibly shared location by a processor cause no bus transactions, as long as no other processor accesses that location. Assuming sufficiently large private caches, cache coherency traffic is the dominant component of bus traffic [17]. Therefore, the bus traffic can be assumed to consist entirely of the communication of partial results[1] between concurrently operating processors. The objective of our partitioning scheme is to minimize cache coherency traffic subject to the constraint of allocating equal sized tasks to each processor for load balancing.

B. Program Model

This paper focuses on *iterative parallel loops*. These loops are defined as two nests of loops and a code body. The *outer nest* consists of sequential loops. The *inner nest* is a pair of parallel loops, each traversing an *iteration set*. For the inner loops, the *iteration space* I is defined as $I = I_1 \times I_2$. Each iteration of the outer sequential loop is a *cycle*. The *code body* can be any set of B executable statements, so long as a mapping from the data sets of the program onto the iteration space is given. The only restrictions placed on the code body are that the computation and the access pattern are identical at each point in I.

For this paper, the code body has been restricted to an update of a data point $A(i, j)$ within a single, two-dimensional global matrix A. Any code body may be used, but this example greatly simplifies the mapping of the data set onto the iteration space. For any memory reference to the global matrix within the right-hand side of the data point update of the code body, the reference will be of the form $A(i + a, j + b)$, where a and b are integer constants. The ordered pair $\boldsymbol{q} = (a, b)$ is known as the *access vector*. The set of all access vectors for the code body is known as the *stencil set*.

Consider the example iterative parallel loop presented in Fig. 1. The iteration set for each of the inner loops is $[1 \ldots n]$. The iteration space for the nest of inner loops is $[1 \ldots n] \times [1 \ldots n]$. The code body is the statement that updates $A(i, j)$. For this code body, the set of access vectors for right-hand side accesses to the array {(2, 0), (1, 0), (−1, 0), (−2, 0), (0, 1), (0, −1)} is the stencil set, and each component of the stencil set is an access vector.

This program construct was chosen because of the following factors:

- These constructs are likely to appear in numerical application programs, e.g., asynchronous solutions to partial differential equations [13], continuum modeling [18], and image smoothing [19].
- The updating of each point within the inner loop iteration space is an independent operation, so parallelism in the problem is much greater than the parallelism inherent in a multiprocessor supporting medium-grained parallelism.
- The outer sequential DO loop requires that the values used to update each data point are those computed in the previous cycle. Therefore, the problem has communication that can significantly impact performance.

An integral part of this work is the ability to optimize in the presence of asymmetric communication. Machine-dependent effects, such as the cache line size effects, can impose nonuniform communication on a symmetric stencil. However, asymmetric stencils do occur in some codes. For example, in some image smoothing code [19], the stencil

$$S = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\} \tag{3.1}$$

is used to compute x and y partial derivatives of boundary motions. As we will see later, an irregular hexagon reduces communication for this particular stencil.

As defined in [20], two types of interprocessor communication are identified, i.e., data communication and control communication. Data communication occurs when processors exchange data required for computation. Control communication occurs when processors exchange control information to accomplish tasks (such as synchronizing at a barrier or semaphore updates). Since C_d, the data communication time, scales with the problem size, C_d accounts for the bulk of the communication penalty paid for reasonably large problems. In a static partitioning, C_c, the control communication time, is a function of the number of processors and is independent of the partition. Therefore, partitions will be evaluated solely on their data communication time.

IV. Quantifying Communication

A method to quantify communication patterns within a

[1] These are points which are computed by one processor and read by another processor.

program is necessary to construct data partitions that minimize communication. This section provides constraints on partition shapes and identifies features of the stencil set that can be used to quantify communication patterns.

A. Partitions

Valid partitions are required to satisfy the following conditions similar to [9]. Practical partitionings such as rectangles and squares satisfy these properties.

1) Tessellation. The partition must tile the iteration space so that each point in the iteration space is assigned to exactly one processor. Given n^2, the size of the iteration space and p, the number of processors, s, the number of data points per part is approximately,
$$s = \frac{n^2}{p}. \tag{4.1}$$
2) Identical by translation. Each partition should be the image of any other one by a translation, except for parts located near boundaries of the iteration space. This property ensures reasonable load balancing and efficient code generation.
3) Straight Line Perimeters. In order to ensure tiling and simplify code generation, the perimeter of each part is composed of line segments, $\{l_i\}$, whose slopes (for non-vertical lines) are rational numbers. The last requirement is to ensure that line segments do not pass through points in the discretized iteration space leading to "kinks."
4) Symmetry. Line segments $\{l_i\}$ can be paired so that segments in each pair have the same orientation and length, i.e., $\{l_i, l_j\}$ such that $|l_i| = |l_j|$ and l_i is parallel to l_j.

B. Generating Communication Weights

1) Target Sets and Shadows: An *access arc* for an access vector $\boldsymbol{q} = (a, b)$ is a vector directed from an array location (i, j), to $(i+a, j+b)$. The target set of a data point y under a stencil gives all data points required for the computation of y.

Definition 1: Let $y = (i, j) \in A$, the global matrix, and $S = \{(a_0, b_0), \cdots, (a_k, b_k)\}$ be the stencil set. Then *the target set of y under S* is $T_S(y) = \{(i+a_0, j+b_0), \cdots, (i+a_k, j+b_k)\}$.

The shadow of a part P, under a stencil S, is the collection of points external to P which are required for the computation of the points of P and its size is equal to the number of points that have to be brought in from other processors in order to update the data points in P.

For a given part P and stencil set S, the *shadow of P under S* is $P_S = \{x | x \notin P \text{ and } \exists y \in P \text{ s.t. } x \in T_S(y)\}$.

2) A Single Access Vector: Consider the communication incurred by a line segment pair, $\{l_i, l_j\}$ with a length L due to an access vector of magnitude q oriented at an angle θ with respect to $\{l_i, l_j\}$. The communication incurred by $\{l_i, l_j\}$ is defined to be the number of accesses that cross l_i or l_j from within the part P. The communication weight of a line segment $\{l_i, l_j\}$ is the communication incurred per unit length and is equal to the increase in communication resulting from a unit increase in both l_i and l_j.

The *communication weight per unit length* is the number of accesses per cycle outside of a given part across a unit length of a line segment pair constituting a part of a perimeter.

Theorem 1: The communication incurred by $\{l_i, l_j\}$ is $Lq\sin\theta$ and the communication weight per unit length for $\{l_i, l_j\}$ is $q\sin\theta$.

Proof: Without loss of generality, assume that the head of an access vector for a point located on the boundary of l_i lies outside the part. The number of points accessed by the elements in the part across l_i consists of the number of points in a rectangle of dimensions L and $q\sin\theta$, as illustrated in Fig. 2. The statement of the theorem follows from the definition of communication weight per unit length. □

For example, consider the access arcs associated with the access vector (3,2) (Fig. 3). The decomposition $q\sin\theta$ along the horizontal axis gives the amount of extra communication incurred for every data point along a vertical partition border, which is two. The decomposition along the vertical axis gives the amount of extra communication incurred for every data point along a horizontal border.

Given an access vector and an arbitrary partition perimeter of m pairs of line segments of length $l_i, 1 \le i \le m$, Theorem 2 gives the communication incurred per cycle by each part.

Theorem 2: Assume an access vector $\boldsymbol{q}$ of magnitude q, and an arbitrary partition perimeter of m pairs of line segments, each of length $l_i, 1 \le i \le m$. Define $n_i = q\sin\theta_i$, for all $1 \le i \le m$. The communication, C_p, incurred on each cycle is approximately
$$C_p = \sum_{i=1}^{m} l_i n_i. \tag{4.2}$$

Proof: Each pair of line segments accounts for $n_i l_i$ communication by Theorem 1. Therefore, the total communication for all m line segment pairs is as above. However, some data points near the corners of the part may be counted twice in this analysis, hence the approximate nature of the result. However, the term representing corner points is only a function of the access vectors and independent of the dimensions of the part. □

Consider a rectangular partitioning scheme, where each rectangle has dimensions h and v along the horizontal and vertical directions. As shown in Fig. 4, the associated communication weights defined to be $n_v = n_1$ and $n_h = n_2$ are the first and second elements of the access vector. The approximate communication as determined by Theorem 2, C_p, is
$$C_p = hn_h + vn_v. \tag{4.3}$$

For hexagons, a similar expression of C_p is possible in terms of the horizontal, back-diagonal[2], and diagonal[3] orientations. There are three distinct line segment pairs, whose lengths are denoted h, b, and d for horizontal, diagonal, and back-diagonal distances. The equation for C_p, with n_h, n_b, n_d defined analogously to n_v and n_h above, is
$$C_p = hn_h + bn_b + dn_d. \tag{4.4}$$

[2]The upper left-hand and the lower right-hand sides

[3]The lower left-hand and upper right-hand sides

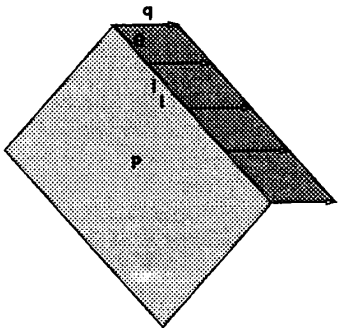


Fig. 2. Communication along a line segment.

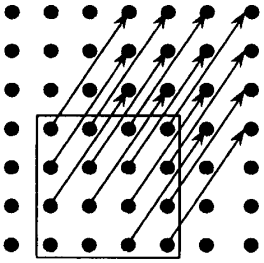


Fig. 3. Access pattern for the stencil set ((3,2)).

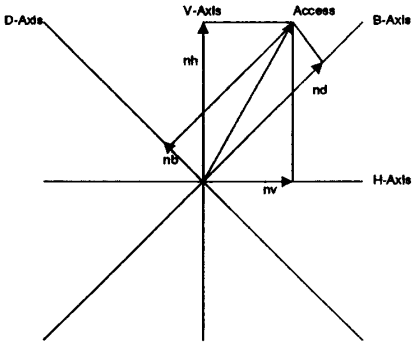


Fig. 4 Decomposing a vector along two separate basis sets of vectors.

3) Multiple Access Vectors: Having developed a method for determining communication weights for a single access vector, we now turn to a set of k access vectors, $\{\boldsymbol{q}_1, \boldsymbol{q}_2, \cdots, \boldsymbol{q}_k\}$. Using the method presented in the previous section, we can generate $N_h = \{n_{h1}, n_{h2}, \cdots, n_{hk}\}$ and $N_v = \{n_{v1}, n_{v2}, \cdots, n_{vk}\}$. Two methods are presented to generate n_h and n_v, composite communication weights, from N_h and N_v.

The major consideration for accuracy is referencing the same data point twice. For example, consider the stencil set ((1, 3), (1,2)). For some point $A(i,j)$, assume the first access vector requires that $A(i+1, j+3)$ be fetched for updating purposes. Assume $A(i, j+1)$ is also within the same part as $A(i,j)$, and the second access vector requires that $A(i+1,(j+1)+2)$, which is really $A(i+1, j+3)$, be fetched as well.

The first method treats all accesses as distinct and is called *additive construction.* The procedure assumes that every access vector contributes to bus traffic. The additive construction gives an upper bound on the coherence traffic.

Theorem 3: Given k access vectors $S = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \cdots, \boldsymbol{q}_k\}$, with horizontal communication components $\{n_{h1}, n_{h2}, \ldots, n_{hk}\}$ and vertical communication components, $\{n_{v1}, n_{v2}, \cdots, n_{vk}\}$, the communication weight per orientation using additive construction is given by,

$$n_h = \sum_{i=1}^{k} |n_{hi}| \text{ and } n_v = \sum_{i=1}^{k} |n_{vi}| \tag{4.5}$$

Proof: For each $\boldsymbol{q}_i \in \{\boldsymbol{q}_1, \boldsymbol{q}_2, \cdots, \boldsymbol{q}_k\}$, $|P_{qi}|$ data points need to be transferred to the processor. Observe that, for an arbitrary rectangle of width h and height v,

$$|P_{qi}| = h|n_{hi}| + v|n_{vi}| - |n_{vi}||n_{hi}|.$$

Therefore, the total amount of communication induced by the stencil is

$$\sum_{i=1}^{k} |P_{qi}| = h\sum_{i=1}^{k} |n_{hi}| + v\sum_{i=1}^{k} |n_{vi}| - \sum_{i=1}^{k} |n_{hi}||n_{vi}|.$$

And, since n_h is the communication weight associated with the horizontal border, $n_h = \sum_{i=1}^{k} |n_{hi}|$. Similarly, $n_v = \sum_{i=1}^{k} |n_{vi}|$, and so the claim is established. □

In contrast, the second method, called *max-min construction,* assumes best case behavior. Any external point need only be fetched once to satisfy all of its references within a particular cycle of the loop. In this case, only the largest communication weight in any given direction contributes to the load on the system bus.

Theorem 4: Given k access vectors $S = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \cdots, \boldsymbol{q}_k\}$, with horizontal communication components $\{n_{h1}, n_{h2}, \cdots, n_{hk}\}$ and vertical communication components $\{n_{v1}, n_{v2}, \cdots, n_{vk}\}$, define,

$$n_h^+ = \max\left(\{n_{hi} | n_{hi} \geq 0\} \bigcup \{0\}\right)$$
$$n_v^+ = \max\left(\{n_{vi} | n_{vi} \geq 0\} \bigcup \{0\}\right)$$
$$n_h^- = |\min\left(\{n_{hi} | n_{hi} < 0\} \bigcup \{0\}\right)|$$
$$n_v^- = |\min\left(\{n_{vi} | n_{vi} < 0\} \bigcup \{0\}\right)|.$$

The communication weight per orientation is given by

$$n_h = n_h^+ + n_h^- \text{ and } n_v = n_v^+ + n_v^-.$$

Proof: From the definition of P_S, the quantities n_h^+, n_h^-, n_v^+, n_v^- and the construction of the rectangle $EFGH$ in Fig. 5, it follows that $P_S \subset EFGH$ and $P_S \bigcap ABCD = \emptyset$. Therefore, $P_S \subset (EFGH \setminus ABCD)$. This immediately implies,

$$\begin{aligned} |P_S| &\leq |EFGH| - |ABCD| \\ &= (n_v^- + h + n_v^+)(n_h^- + v + n_h^+) - hv \\ &= h(n_h^+ + n_h^-) + v(n_v^+ + n_v^-) + E \end{aligned}$$

where

$$E = n_h^+ n_v^+ + n_h^+ n_v^- + n_h^- n_v^+ + n_h^- n_v^-.$$

Let q_i, q_j be the access vectors such that their horizontal components are n_h^+, n_h^-, and let P_{qi} and P_{qj} denote their shadows. Then, $P_S \bigcap IJGH \supseteq P_{qi} \bigcap IJGH$, since $P_{qi} \subseteq P_S$. This implies $|P_S \bigcap IJGH| \geq |P_{qi} \bigcap IJGH| = hn_h^+$. Similarly, $|P_S \bigcap EFLK| \geq hn_h^-$.

Similarly, considering the access vectors that have vertical components of n_v^+ and n_v^-, we can show

$$|P_S \bigcap BLJC| \geq vn_v^+ - n_v^+(n_h^+ + n_h^-)$$
$$|P_S \bigcap KADI| \geq vn_v^- - n_v^-(n_h^+ + n_h^-).$$

Summing up the four equations,

$$|P_S| \geq (n_h^+ + n_h^-)h + (n_v^+ + n_v^-)v - E.$$

From this is obtained

$$(n_h^+ + n_h^-)h + (n_v^+ + n_v^-)v - E$$
$$\leq |P_S| \leq (n_h^+ + n_h^-)h + (n_v^+ + n_v^-)v + E.$$

Since n_h is the communication weight associated with the horizontal border, $n_h = n_h^+ + n_h^-$. Similarly, $n_v = n_v^+ + n_v^-$, and so the claim is established. □

In both construction procedures, the communication $C_p = hn_h + vn_v + \mathcal{E}$, where $\mathcal{E}$ is an error term. Since in either case the constant error term is independent of the dimensions of the rectangle, the magnitude of $\mathcal{E}$ has no bearing on any optimization procedure. Hence, we will ignore $\mathcal{E}$ in the rest of this paper. Additive constructions are applicable when the probability that a data point is invalidated between two references of that point is high. This probability depends on the ratio of shared to nonshared data. For our applications, the granularity of the tasks is coarse, so the ratio of shared to nonshared data is small. For this reason, the max-min construction will be used when experimental results are discussed.

4) Hexagonal Communication Weights: Determining communication weights per unit length for a hexagonal partition is an extension of rectangular decompositions. For a single access vector, an analysis similar to rectangles is done for communication weights in terms of h, b, and d the horizontal, back-diagonal, and diagonal dimensions as in Fig. 4. The only difference is decomposing a vector along *two* separate basis sets of vectors in order to observe communication in the horizontal/vertical vector space, as well as the diagonal/back-diagonal vector space. These decompositions are referred to as n_v, n_h, n_d, and n_b.

Previous algorithmic work on hexagons used regular hexagons with 60 degree exterior angles. In a discretized plane, a line segment at a 60 degree angle to the horizontal passes through data points. In order to maintain symmetry and tessellation constraints, jagged nonhorizontal line segments were introduced at the expense of considerable run-time computational overhead. Such hexagons do not satisfy the condition of straight line perimeters. Instead, the hexagons implemented in this paper have a 45 degree exterior angle between horizontal and nonhorizontal segments. These hexagons have lower run-time overhead. However, a 45 degree regular hexagon has a slightly higher perimeter-to-area ratio than a 60 degree regular hexagon and incurs a larger communication overhead. The B-Axis and D-Axis lie at 45 degree angles to the H-Axis and V-Axis due to the use of 45 degree regular hexagons.

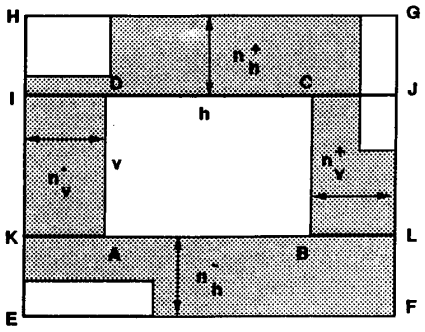


Fig. 5. An analysis of communication patterns.

For a full stencil set of an arbitrary number of access vectors, the previous construction methods are retained. All of the individual decompositions are taken. The max-min construction is applied to generate the communication weight per unit length for a partition boundary with that orientation.

V. Compensation for Cache Line Size

The communication weights derived in the previous section measure the number of data points that are exchanged across a unit length of a partition perimeter. However, in most multiprocessor systems, the basic unit of intercache or bus transfer is an entire cache line. Therefore, the communication weight must be scaled so that the unit of measure is the number of cache lines transferred per unit length of partition perimeter. Such scaling affects the relative magnitude of communication in various directions and hence the shape of the partition. In the following we assume column-major storage and a cache line size of l data points.

A. Compensation for Rectangles

In this subsection, we determine the number of cache lines transferred across unit length of partition perimeter given the corresponding number of data points and the cache line size.

Theorem 5: Given n_h, n_v, and l, the number of cache lines required per unit length along a horizontal boundary, n_h^c, is given by

$$n_h^c = \left\lceil \frac{n_h}{l} \right\rceil \tag{5.1}$$

when the cache lines and horizontal partition borders are aligned and by

$$n_h^c = \left\lceil \frac{n_h}{l} \right\rceil + \frac{(l-1) - \left(\left\lceil \frac{n_h}{l} \right\rceil l - n_h\right)}{l} \tag{5.2}$$

when cache lines are skewed with respect to the horizontal partition borders. The number of cache lines required per unit

length along the vertical boundary, n_v^c, is

$$n_v^c = \frac{n_v}{l} \tag{5.3}$$

Proof: Two cases are identified and n_h^c is derived for each case. In the first case, the partition boundary is assumed to be aligned with the cache line boundaries (Fig. 6). This case occurs when array column dimensions and v are chosen to be multiples of the cache line size. Examining the shadow of an access vector across a horizontally-aligned border, the size of the shadow is given by (5.1).

In the second case, cache line boundaries are randomly placed with respect to the horizontal boundary (Fig. 7). This case occurs when the column dimension and v are mutually prime. In such a case, n_h^c is composed of two terms, the first being identical to (5.1), and the second being a correction term which is always less than one. Consider a grouping of l consecutive columns, each with a unique skew, s, in the range $0, \cdots, l-1$. The cache lines required by a column with skew s is

$$\left\lceil \frac{n_h + s}{l} \right\rceil.$$

And observe that

$$\left\lceil \frac{n_h + s}{l} \right\rceil = \begin{cases} \left\lceil \frac{n_h}{l} \right\rceil & \text{if } \left\lceil \frac{n_h}{l} \right\rceil l \geq n_h + s \\ \left\lceil \frac{n_h}{l} \right\rceil + 1 & \text{otherwise} \end{cases}$$

$$l - 1 \geq s > \left\lceil \frac{n_h}{l} \right\rceil l - n_h.$$

So on the average,

$$\frac{(l-1) - \left(\left\lceil \frac{n_h}{l} \right\rceil l - n_h\right)}{l}$$

extra cache lines are required. Hence, the result.

Now, considering communication across a vertical border, (or any border whose access vectors are orthogonal to the orientation of the cache lines), l data points along the border are in the same cache line, so in reality only one transfer is required per l points. Therefore, (5.3) describes the communication. □

Equations (5.1) and (5.3) yield similar ratios for n_h^c to n_v^c for large values of n_h and n_v. However, if n_h and n_v are much smaller than l (as is usually the case), n_h^c and n_v^c are quite different, e.g., $n_h = n_v = 1$, $l = 8$, implies $n_h^c = 1$, $n_v^c = 0.125$.

B. Compensation for Hexagons

When considering hexagonal partitions, n_h^c is given by formula (5.1) or (5.2). The b and d borders make a 45 degree angle with the cache lines and we now consider the effect aligned cache lines have on the b border.

Theorem 6: Given n_b, the average number of cache lines required across the b border is

$$n_b^c = \frac{1}{\sqrt{2}}\left[\left\lceil \frac{n_b\sqrt{2}}{l} \right\rceil + \frac{(l-1) - \left(\left\lceil \frac{n_b\sqrt{2}}{l} \right\rceil l - n_b\sqrt{2}\right)}{l}\right] \tag{5.4}$$

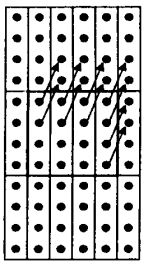


Fig. 6. Cache lines aligning with borders.

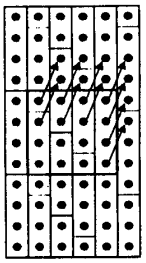


Fig. 7. Cache lines randomly placed with respect to borders.

where l is the cache linewidth in terms of the number of data points, and

Proof: Observe that, for each distance of $\sqrt{2}$ along the horizontal direction, that the number of points communicated across the b border is $\sqrt{2}n_b$. From this, $\sqrt{2}n_b$ can be substituted for n_h in (5.2). □

A formula recasting n_d^c in terms of n_d is similar to equation (5.4).

VI. Partition Construction

In this section, the communication parameters are used to determine the optimal partitioning from a class of partitions. The superscript "c" is dropped from the communication parameters in this section to simplify the notation. However, all communication parameters are to be compensated for cache line size.

A. Rectangular Partitioning

Theorem 7: Let S be a stencil set yielding n_h and n_v as communication weights per orientation. Then, C_p is minimized for rectangles when

$$\frac{h_o}{v_o} = \frac{n_v}{n_h}. \tag{6.1}$$

Proof: C_p is given for a rectangular partition by (4.3). The number of data points per processor, s, is given in (4.1) and for rectangles is $hv = s$. Using this to substitute for s in (4.3), C_p is now a function of h. Then, as seen by taking the partial derivative with respect to h, and assuming that $s \gg hn_h + vn_v$, communication time is minimized by (6.1). And so, the claim is shown. □

Therefore, given the communication in the horizontal and vertical directions and the number of data points per partition, (6.1) gives the border lengths for a rectangular partition that minimizes interprocessor communication. The theorem states that the communication for a partition is minimized when the communication across each boundary is equalized. A similar result was obtained for clustered multiprocessors in [21]. As an interesting special case, consider the case $n_h = n_v$. The aspect ratio, $h/v = 1$, which indicates a square partitioning. The same results were found experimentally in [14] and demonstrated formally in [16].

B. Hexagonal Partitioning

Theorem 8: Let S be a stencil set yielding n_h, n_b, and n_d as communication weights per orientation. Then, C_p is minimized for hexagons where

$$b_o = \frac{\sqrt{s}(\sqrt{2}n_h - n_b + n_d)}{\Delta} \tag{6.2}$$

$$d_o = \frac{\sqrt{s}(\sqrt{2}n_h - n_d + n_b)}{\Delta} \tag{6.3}$$

$$h_o = \frac{\sqrt{s}(\sqrt{2}n_b + \sqrt{2}n_d - 2n_h)}{\Delta} \tag{6.4}$$

and

$$\Delta = \sqrt{2\sqrt{2}n_h(n_b + n_d) - n_b^2 + 2n_b n_d - n_d^2 - 2n_h^2}.$$

Proof: The same method as the previous theorem is used [22]. □

Consider the special case where $n_h = n_b = n_d$. This directly implies from (6.2), (6.3), and (6.4) that a hexagon where $h_o = b_o = d_o$ has the minimum associated communication. This result was previously demonstrated both formally and experimentally in [16], and once again shows how previous algorithmic partition information can be explained as a special case of ADP.[4] A multidimensional extension to both the rectangle and hexagon optimizations has been developed [22], but is omitted here due to lack of space.

VII. Experimental Evaluation of ADP

ADAPT (Automatic Data Allocation and Partitioning Tool), a program partitioner implementing ADP, was written to translate a parallel Fortran program into a program implementing a desired partition. ADAPT is a partial evaluator that takes as input an annotated Fortran program and parameter values such as the cache line size as input and generates a syntactically correct Fortran program complete with parallelizing instructions specific to the Multimax. Currently, ADAPT partitions program segments with a single outer sequential loop and two inner parallel loops. ADAPT determines the stencil set from the code body, generates communication weights, and partitions the program.

[4]Previous results were for a 60 degree regular hexagon, not a 45 degree regular hexagon. A similar treatment for 60 degree regular hexagons is possible.

A. The Sample Problem

The code segment given in Fig. 1 was used for the experiments. Its stencil has a nonuniform communication pattern. Therefore, standard geometric partitions, as proposed in [13], [16], and [14], fail to minimize communication. Also, this application contains no independent execution sets and analytical partitioning schemes, as presented in [2] and [3], fail to identify communication-reducing partitions. Timings of code conducted on 16 processors of a 20-processor Encore Multimax at Argonne National Laboratory show that a performance gain was attained by ADP.

In all of the experiments and simulations, the size of the part assigned to each processor was chosen to be less than the cache size to eliminate compulsory cache misses. The array dimensions and partition placements were chosen to reduce conflict misses. The various times for various partitions therefore reflect as accurately as possible the differing levels of cache coherency traffic.

Two different sets of experimental evaluations were conducted to demonstrate the utility of ADP. First, the execution times of parallel programs partitioned using ADP and using conventional approaches are measured on 16 processors of an Encore Multimax. Second, programs partitioned using competing approaches are simulated on a multiprocessor simulator to measure miss ratios. The simulation results offer insight into how the partitioning strategy works.

B. Execution Results

The decomposition of each access vector is given in Table I. Using the max-min construction method, $n_h = 4$, and $n_v = 2$. On the Multimax, $l = 4$, since the cache line size is 16 bytes, and the size of a floating-point number is 4 bytes. Since the array dimensions are multiples of l, the communication weights are adjusted using (5.1) and (5.3) to $n_h^c = 1$ and $n_v^c = 0.5$ to accommodate the cache line size effect. From (6.1), the aspect ratio[5] of the optimal rectangular partition is 0.5.

The performance of standard dynamic and static partitions will be compared to that of the rectangle generated by ADAPT. The Guided Self-Scheduling Method [20] is used as an example of a dynamic program partition.[6] Standard static program partitions (rows and squares) are compared to partitions generated using ADP.[7] The rectangles designed by ADAPT improved execution time by 8.2% over squares, 9.1% over rows, and 15.2% over guided self-scheduling.

Experiments were conducted on the Encore Multimax using rectangular partitions of equal area and varying aspect ratio. These results are presented in Fig. 9, in which the average execution time in milliseconds is plotted against the aspect ratio. Clearly, partitions with width to length ratios close to 0.5 minimize execution time for this example, thus demonstrating the accuracy of the max-min construction algorithm and (5.1) for the program segment in Fig. 1 on the Encore Multimax.

[5]The ratio of h to v

[6]GSS would be more applicable if dynamic load balancing were required.

[7]Fig. 8 gives the execution rates in units of KFLOPS from test cases run on 16-processors of an Encore Multimax.

TABLE I
VECTOR DECOMPOSITIONS FOR 6-POINT STENCIL

Vector	n_{hi}	n_{vi}	n_{bi}	n_{di}
1	2	0	$\sqrt{2}$	$\sqrt{2}$
2	1	0	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{2}}{2}$
3	-1	0	$-\frac{\sqrt{2}}{2}$	$-\frac{\sqrt{2}}{2}$
4	-2	0	$-\sqrt{2}$	$-\sqrt{2}$
5	0	1	$\frac{\sqrt{2}}{2}$	$-\frac{\sqrt{2}}{2}$
6	0	-1	$-\frac{\sqrt{2}}{2}$	$\frac{\sqrt{2}}{2}$
Component	4	2	$2\sqrt{2}$	$2\sqrt{2}$

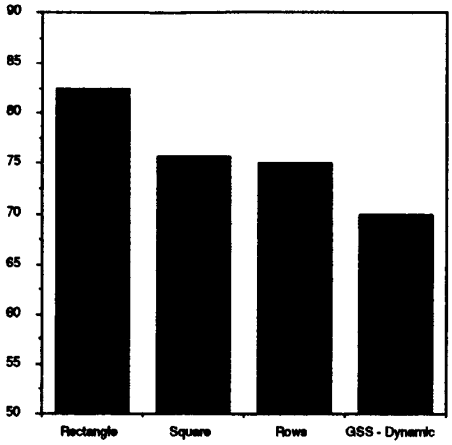


Fig. 8. Execution rates for various partitions.

The low FLOPS numbers are due in part to the comparatively modest floating-point performance of the Encore Multimax (0.5 MFLOPS/processor peak performance).

C. Simulation Results

1) Varying Cache Line Sizes: In order to experimentally verify the effect cache line sizes could have on communication, the program segment from Fig. 1 was run on a simulator of a multiprocessor configuration of the Astronautics ZS-1. Each processor has a private cache and is connected to main memory via a crossbar. Cache coherence is maintained through the use of a directory-based scheme. The code segment was run using partitions of varying inverse aspect ratios while keeping the overall partition size constant to determine which partition shapes minimized the miss ratio. In addition, this experiment was repeated for various values of l, the number of data points per cache line. The results are summarized in Fig. 10.

From an analysis of the code it is clear that $n_h = 4$ and $n_v = 2$. Given this, a machine-independent optimizer using (6.1) generates a partition with an inverse aspect ratio of two to one. However, a machine-dependent partitioner uses (5.2) and (5.3) yielding different partitions for various values of l as given in Table II. A comparison of the simulated miss ratios for the partitions generated using machine-independent weights and the machine-dependent weights is given in Table III. The machine-dependent partitioner generates a partition whose performance is close to the partition that achieves the minimum miss ratio for a particular line size.

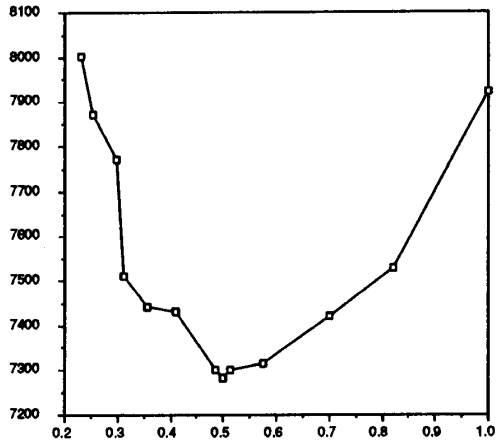


Fig. 9. Execution time of partitions on Multimax.

2) Simulation of Hexagonal Partitions: A hexagonal partition for a particular line size must consider the orientation of the hexagons. The hexagons may be oriented horizontally, or they may be oriented vertically, as obtained by rotating a hexagon by 90 degrees. For a horizontally oriented hexagon, C_p, is given in (4.4). Similarly, for a vertically oriented hexagon,

$$C_p = vn_v + dn_d + bn_b \tag{7.1}$$

and must be minimized.

We use the code segment given in Fig. 1. Three partitions were simulated using the Astronautics ZS-1 multiprocessor simulator using a line size of 16 bytes. Solving (6.4), (6.2), and (6.3) for a horizontally oriented hexagon yields $h_o = 0$, $b_o = d_o = 25$, and $C_p = 141.4$. This partition is a square rotated 45 degrees. The simulation of the program partitioned by these hexagons has a miss ratio of 0.62%. For the vertically oriented hexagons, $h_o \approx 20$, $b_o = d_o \approx 14$, and $C_p = 119.20$, from (7.1). The simulation of the program partitioned by these hexagons has a miss ratio of 0.76%. Both of the previously computed values compare favorably to a regular hexagon, which has $h = 16$, $b = d = 17$, and $C_p = 160.16$. The simulation of the program partitioned by regular hexagons has a miss ratio of 1.31%. Note that the use of either of the optimized hexagons results in a near halving of the miss ratio compared to using regular hexagons.

3) Simulation of Partitions for a Nonuniform Stencil: In some image smoothing code [19], the stencil (3.1) is used to compute x and y partial derivatives of boundary motions. This stencil has uniform communication across the horizontal and vertical boundaries, while having unequal communication across the b and d boundaries. This stencil was simulated using

TABLE II
THE OPTIMAL INVERSE ASPECT RATIOS FOR VARIOUS LINE SIZES

l	n_h^c	n_v^c	n_h^c/n_v^c
4	1.75	0.5	3.5
8	1.375	0.25	5.5
16	1.1875	0.125	9.5

TABLE III
THE MISS RATIO PERCENTAGES FOR VARIOUS LINE SIZES

l	Independent	Dependent	Improvement	Percentage Improvement
4	2.88	2.82	0.06	2%
8	2.04	1.60	0.44	27.5%
16	1.75	1.23	0.52	42.3%

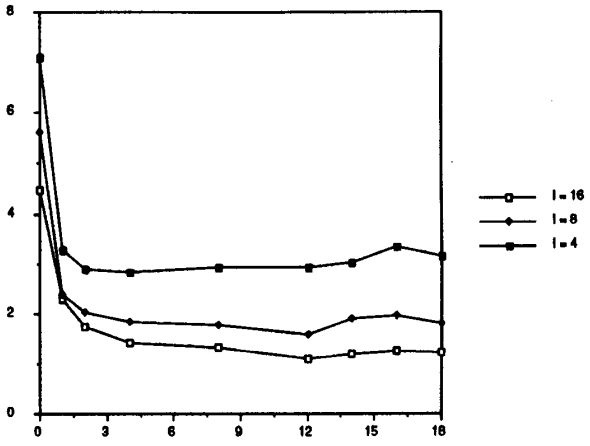


Fig. 10. Percentage miss ratios for various inverse aspect ratios and line sizes.

the Astronautics ZS-1 simulator, using a cache line size of $l = 16$.

For this example, the simulation of the program partitioned by regular hexagons has a miss ratio of 1.93%. The simulation of the program partitioned by vertically optimized hexagons with linewidth consideration, which were generated using (6.2), (6.3), (6.4), (5.2), and (5.4), has a miss ratio of 0.98%. The optimized hexagons improved the miss ratio over the regular hexagons by 49.2%.

An experiment with rectangular partitions was conducted. The simulation of the program partitioned by squares has a miss ratio of 1.56%. The simulation of the program partitioned by optimized rectangles, which were generated using (6.1) and (5.2), has a miss ratio of 0.87%. The optimized rectangles improved the miss ratio over the squares by 44.2%.

The optimized rectangles have a lower miss ratio than optimized hexagons, even though the modeling gives C_p values of 150 and 141.1, respectively. We believe that the discrepancy is because the rectangular partitions were aligned so that no two parts shared the same cache line. Such alignment of optimized hexagons is not possible because of the diagonal boundaries. False sharing has an impact on cache coherence traffic when line sizes are large. But, the effects of false sharing are not accounted for by our model and represents an area of future work.

VIII. CONCLUSION

This paper examined program partitioning as a method for decreasing parallel program execution time. Communication within a restricted iterative parallel loop was quantified by a vector notation. Using this measure of communication, data partitions with minimum interprocessor communication were designed using adaptive data partitioning. ADP lends itself to being a step toward a theoretical framework for analyzing communication in parallel programs executed on shared-memory multiprocessors.

ADP is an optimal partitioning scheme using rectangles and nonregular hexagons for any stencil. This differed significantly from previous work, which focused on development of optimal data partitions for specific stencil sets. In addition, ADP is based on algorithmic solutions which were inspired by scrutiny of common place numerical applications. The result was an automatic analysis that could reduce cache coherency traffic. ADAPT, an automatic data partitioning code restructurer, was used to analyze programs. Experiments on an Encore Multimax and Astronautics ZS-1 simulator demonstrated the advantages of ADP. Both hexagons and rectangles were analyzed. Hexagons, which traditionally have been touted for partitioning, were outperformed by rectangles.

Many opportunities exist for future work in this area. The restriction of a single, matrix update placed on the code body of the iterative parallel loop will be replaced with a general set of statements in order to make ADP applicable to as many programs as possible. Also, ADP can be expanded to other parallel programming constructs and various types of multiprocessor systems. Such a comprehensive analysis will lead to the development of a class of compilers which can minimize communication in portable parallel programs across a wide range of multiprocessors.

ACKNOWLEDGMENT

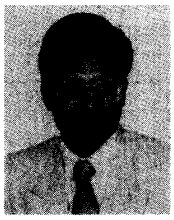
The Mathematical and Computer Science Division at the Argonne National Laboratory provided access to the Encore and Sequent multiprocessor. They also supplied invaluable information and advice in using the machines. R. Clapp provided help using the simulator. The detailed comments of the anonymous reviewers improved the quality of the presentation.

REFERENCES

[1] D. A. Padua, "Multiprocessors: Disscussions of some theoretical and practical problems," Ph.D. dissertation," Univ. Illinois at Urbana–Champaign, Rep UIUCDCS-R-79–99, 1979.

[2] J. Peir and R. Cytron, "Minimum distance: A method for partitioning recurrences for multiprocessors," *IEEE Trans. Comput.*, vol. 38, pp. 1203–1211, Aug. 1988.

[3] W. Shang and J. A. B. Fortes, "Independent partitioning of algorithms with uniform dependencies," in *Proc. Int. Conf. Parallel Processing*, 1987, pp. 26–33.
[4] E. H. D'Hollander, "Partitioning and labeling of index sets in do loops with constant dependence vectors," in *Proc. Int. Conf. Parallel Processing*, 1989, pp. 139–144.
[5] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proc. 3rd SIAM Conf. Parallel Processing Scientific Comput.*, 1987, pp. 357–361.
[6] ——, "More iteration space tiling," in *Proc. Supercomputing '89*, 1989, pp. 655–664.
[7] J. Ramanujam and P. Sadayappan, "Tiling of iteration spaces for multicomputers," in *Proc. Int. Conf. Parallel Processing*, 1990, pp. 23–36.
[8] C. King and L. M. Ni, "Grouping in nested loops for parallel execution on multicomputers," in *Proc. Int. Conf. Parallel Processing*, 1989, pp. 31–38.
[9] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proc. 15th Annu. ACM SIGACTSIGPLAN Symp. Principles Programming Languages*, ACM, 1988, pp. 319–329.
[10] J. L. Hennessey and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Los Altos, CA: Morgan Kaufmann, 1990.
[11] K. Gallivan, W. Jalby, and D. Gannon, "On the problem of optimizing data transfers for complex memory systems," in *Proc. ACM Int. Conf. Supercomput.*, 1988, pp. 238–253.
[12] K. Gallivan, W. Jalby, U. Meier, and A. H. Sameh, "Impact of hierarchical memory systems on linear algebra algorithm design," *Int. J. Supercomput. Appl.*, vol. 2, no. 1, pp. 12–48, Spring 1988.
[13] G. C. Fox and S. W. Otto, "Algorithms for concurrent processors," *Physics Today*, vol. 37, pp. 50–59, May 1984.
[14] D. Vrsalovic, E. F. Gehringer, Z. Z. Segall, and D. P. Sieworek, "The influence of parallel decomposition strategies on the performance of multiprocessor systems," in *Proc. 12th Annu. Int. Symp. Comput. Architecture*, vol. 13, June 1985, pp. 396–405.
[15] J. H. Saltz, V. K. Naik, and D. M. Nicol, "Reduction of the effects of the communication delays in scientific algorithms on message passing MIMD architectures," Tech. Rep. 86–4, NASA Langley Research Center, 1987.
[16] D. A. Reed, L. M. Adams, and M. L. Patrick, "Stencils and problem partitionings: Their influence on the performance of multiple processor systems," *IEEE Trans. Comput.*, vol. C-36, pp. 845–858, July 1987.
[17] S. J. Eggers and R. H. Katz, "The effect of sharing on the cache and bus performance of parallel programs," in *ASPLOS-III*, pp. 257–270, 1989.
[18] H. S. Stone, *High-Performance Computer Architecture.* Reading, MA: Addison-Wesley, 1987.
[19] B. G. Schunk, "Image flow segmentation and estimation by constraint line clustering," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 11, pp. 1010–1027, Oct. 1989.
[20] C. D. Polychronopoulos, "On program restructuring, scheduling, and communication for parallel processor systems," Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, CSRD TR 595, Aug. 1986.
[21] S. G. Abraham, "Reducing interprocessor communication in parallel architectures: System configuration and task assignment," Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, CSRD TR 726, Jan. 1988.
[22] D. E. Hudak and S. G. Abraham, "Multidimension extensions to adaptive data partitioning," Tech. Rep. CSE-TR-85–91, The University of Michigan, 1991.



Santosh G. Abraham (S'83–M'87) received the B.Tech. degree from the Indian Institute of Technology, Bombay, in 1982, the M.S. degree from the State University of New York, Stony Brook, in 1983, and the Ph.D. degree from the University of Illinois, Urbana in 1988, all in electrical engineering.

He is currently an Assistant Professor in the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. From 1984–1987, he was a Research Assistant in the Center for Supercomputing Research and Development at the University of Illinois. His research interests are in the areas of parallel processing, compilation for parallel systems, and computer architecture.

David E. Hudak is a Ph.D. degree student in the Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor, and a Research Assistant in the Advanced Computer Architecture Laboratory. His research interests focus on hardware and software methods for improving the performance of multiprocessors.