JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 15, 368–381 (1992)

Compile-Time Optimization of Near-Neighbor Communication for Scalable Shared-Memory Multiprocessors

DAVID E. HUDAK AND SANTOSH G. ABRAHAM

Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan 48109-2122

Scalable shared-memory multiprocessor systems are typically NUMA (nonuniform memory access) machines, where the exploitation of the memory hierarchy is critical to achieving high performance. Iterative data parallel loops with near-neighbor communication account for many important numerical applications. In such loops, the communication of partial results stresses the memory system performance. In this paper, we develop data placement schemes that minimize communication time where the near-neighbor interaction is determined by a stencil. Under a given loop partition, our compile-time algorithm partitions global data into four classes for each processor, with each class requiring specific consistency maintenance requirements. The ADAPT (Automatic Data Allocation and Partitioning Tool) system was implemented to automatically partition parallel code segments for the BBN TC2000, a scalable shared-memory multiprocessor. ADAPT caches global arrays and maintains data consistency in software through instructions that flush data from private caches. Restructuring of a fluid flow code segment by ADAPT improved performance by a factor of more than 3 on the BBN TC2000. Features in current generation pipelined processors with multiple functional units permit the overlap of memory accesses with computation. Our experiments on the BBN TC2000 show that the degree of overlap is limited by architectural parameters, such as the number of CPU registers. © 1992 Academic Press, Inc.

1. INTRODUCTION

Shared-memory multiprocessors offer a familiar model for programmers. Scalable multiprocessor systems can be classified as nonuniform memory access (NUMA) machines where the memory latency depends on the locations accessed. For instance, an access by a processor in the BBN Butterfly TC 2000 has a latency of 3, 11, or 38 CPU clock cycles depending on whether the location accessed is in the cache, local memory, or remote memory, respectively. Other scalable, shared-memory multiprocessors such as the MIT Alewife and Stanford DASH multiprocessor systems [20] have nonuniform access latencies. The increased latency and reduced bandwidth of global memory have a substantial impact on performance. Restructuring of programs can reduce the number of global memory accesses and dramatically improve performance.

We believe that future multiprocessor systems will have complex memory hierarchies, which cannot be managed effectively by the hardware. Since the portability of parallel programs is an important issue, and since each multiprocessor will have a unique memory hierarchy, the burden of managing the memory hierarchy will fall on the compiler. The development of compile-time schemes to manage local memories is therefore an important research topic.

Our initial work toward a compiler that automatically compiles code to utilize the memory hierarchy of a scalable multiprocessor system is based on the following premises. A large amount of execution time is spent in parallel loops and initial work should focus on such loops. Numerical programs are typically continuum models where each point in a multidimensional space can be updated in parallel but the newly updated values are required in the next time-step. The particular loop construct studied in this paper is used in coding such iterative data parallel programs. Thinking Machines Corporation has recently introduced a specialized compiler to optimize such loops for the Connection Machine [4].

We have developed a theoretical framework for analyzing communication for such loops. In earlier work, we also developed optimal loop partitioning schemes to determine the loop partition that minimizes the number of data points exchanged between processors [1, 16]. In this paper, given the parallel code segment and the loop partition, we divide each global array into four classes for each processor. The *exclusive read–write set* (ERW) may be moved by each processor into the highest level of its memory hierarchy. Consistency must be maintained on the *shared read–exclusive write set* (SREW) and *shared read–no write set* (SRNW). No accesses are made by the processor to the *no read–write set* (NRW). Such a data partition is automatically obtained from a few parameters, viz., communication parameters derived from the code segment and the loop partition.

Results of this analysis can be used with different data placement schemes, e.g., placing the ERW set in the local memories of each processor. These schemes have been implemented and experimental results on the Butterfly TC2000 are reported. Even a simple data partition

0743-7315/92 $5.00

that caches the ERW set was shown to have a factor of 3 improvement in execution time over currently used default data assignments. We focus on minimizing the impact of communication time, first, by choosing the data assignment to minimize the communication time, and second, by overlapping as much of the remaining communication time with computation as possible. The overlap of computation with communication was shown to yield execution time improvements of nearly 10% for some data assignments.

Software systems that partition and manage the consistency of data using messages among processors are currently being developed for distributed memory machines [12, 19]. In addition, partitioning systems that manage shared data are being developed commercially for SIMD machines [4, 18]. The optimization of code for a shared-memory NUMA system requires similar treatment. Toward this end, we have developed the ADAPT (Automatic Data Allocation and Partitioning Tool) system to analyze the Fortran versions of the iterative parallel loops described in this paper and to generate code for the BBN TC2000 which employs a proper data partition and exploits the memory hierarchy.

2. RELATED WORK

Our goal is to develop compiler techniques for restructuring parallel loops for shared memory multiprocessors to minimize the performance degradation due to the latency and available bandwidth of the memory system. Related work includes work on loop partitioning, automatic data distribution for nonshared memory machines, locality enhancement, prefetching, and software consistency maintenance.

Loop partitioning for two-dimensional iteration spaces is often achieved by tiling the iteration space with geometric shapes that tessellate, as described by Reed *et al.* [26] and Carr and Kennedy [7]. In contrast, this paper discusses a systematic method for reducing the impact of communication time on the execution time of any stencil-based, iterative data parallel loop by generating data placement strategies and overlapping communication and computation. Ramanujam and Sadayappan [25] and Tseng [28] present dependence-oriented partitioning approaches for iteration spaces to be executed on message-passing, nonshared memory multiprocessors. In contrast, our work is oriented toward scalable shared-memory multiprocessors.

Research on the automatic distribution of data has been done for message-passing, nonshared memory systems, e.g., Zima *et al.* [30], Pingali and Rogers [23], and Fortran-D [12]. Our analysis uses an existing process decomposition (ADP) [1, 16] that minimizes data exchange between processors, and then determines the data placement which minimizes communication time. Thus, our system automates and optimizes all the steps involved in mapping an iterative parallel loop to a particular shared-memory multiprocessor system.

Performance improvement through exploitation of memory hierarchies has been previously studied by Gannon *et al.* [14], Carr and Kennedy [8], and Wolf and Lam [29]. These optimizations attempt to maximize reuse of cache data in the context of a finite cache, i.e., minimizing uniprocessor misses. They have a secondary effect of improving multiprocessor performance by reducing the bandwidth requirements of each processor, thereby reducing contention in the memory system. In contrast, our work considers a multiprocessor environment where each processor has its own memory hierarchy.

In prefetching, the data are brought to a higher level in the memory hierarchy before they are required. Since computation continues during the prefetch, the global memory access latency is hidden; see, e.g., Gannon *et al.* [14], Gornish *et al.* [15], and Callahan *et al.* [6]. However, prefetching only helps to hide large memory latencies, and does not reduce bandwidth requirements. Our work is based on multiprocessors that lack explicit support for prefetching. However, we exploit the parallelism between the access unit and the floating point unit in the Motorola 88000 processor [22] to overlap the latency of the remote memory with useful computation. Unlike earlier analytic or simulation work, we measure actual execution times to illustrate the achievable overlap of communication and computation. Automatic compile-time maintenance of consistency for shared data has been examined by Cheong and Veidenbaum [9] and by Cytron *et al.* [10].

3. MODEL ASSUMPTIONS

3.1. Multiprocessor Model

The data placement and overlap strategies discussed here are applicable to a wide range of shared-memory architectures with a scalable memory hierarchy. We assume a multiprocessor with a globally shared address space across physically distributed memories. Since current methods for hardware-maintained cache consistency do not scale well to large numbers of processors, our model multiprocessor assumes no hardware support for cache consistency. We assume a three-level memory hierarchy consisting of a private cache, private local memory, and globally shared remote memory. Note that the effects of contention for shared communication media (memory banks, switches, etc.) are not considered in this work since the focus is on the minimization of overall communication requirements, and not on optimization for any particular interconnection configuration.

The BBN TC2000 Butterfly parallel processor [2] is a

shared-memory MIMD machine composed of Motorola 88000 processors, each having an individual cache. The processor is resident on a *function board* with a memory module. The function boards are interconnected by a multistage interconnection network. The BBN TC2000 is a nonuniform memory access machine. From the perspective of a particular processor on the TC2000, the memory hierarchy is that processor's cache, local memory (i.e., the region of shared memory addresses which correspond to the memory module which is resident on the function board), and global memory (i.e., other processor's local memories). Other scalable shared-memory machines have similar memory hierarchies [20] and conform to our model, but substitute a mesh for the multistage interconnection network. The latencies for various memory operations as determined by BBN [2] are 0.15 μs or 3 CPU cycles for a cache access, 0.60 μs or 11 CPU cycles for a local memory access, and 1.89 μs or 38 CPU cycles for a global memory access.

3.2. Program Model

The iterative data parallel loops analyzed in this paper are a collection of perfectly nested loops with an outermost sequential loop that controls the execution of inner data parallel loops. For simplicity, a two-dimensional parallel iteration space of size $N \times N$ is assumed, although many of our methods generalize to higher dimensions [17]. We assume that the bounds of the inner parallel loop are large enough to warrant parallel execution. A *cycle* is an execution of a single iteration of the outer sequential loop. The body of the loop consists of a series of assignment statements involving two-dimensional array variables. Our analysis assumes that the code body honors the *single assignment* property, i.e., that each array location is written to by only one iteration of the parallel loops. Additionally, we assume asynchronous semantics for the updating of arrays within the parallel loops.

This work develops data storage techniques that optimize near-neighbor communication. Typically, regular near-neighbor communication is expressed in applications through the use of subscript offsets, as in Fig. 1, and irregular communication is characterized by maintaining an array of pointers. Such arrays have been analyzed in substructuring methods for finite element domains [11]. Since such run-time information is not amenable to compile-time analysis, we focus on subscript offsets. Such offsets appear in many numerical application programs, e.g., asynchronous solutions to partial differential equations, continuum modeling, and image smoothing [1]. Therefore, the subscript expressions for right-hand side references of an array are restricted to one of the parallel loop indices plus or minus a small constant. The ordering of the appearance of loop indices is assumed to be identical for all array references. For simplicity, the subscript expressions for left-hand side references of an array are restricted to parallel loop indices. Different types of subscript expressions, such as those found in Gaussian elimination, induce different types of communication, thus requiring further analysis. This is an avenue of important future work.

```
do k = 1, 1000
   do j = 2, N-1
      do i = 2, N-1
         A(i,j) =(B(i+1,j)+B(i-1,j)+B(i+1,j+1)+B(i,j-1))*0.25
      enddo
   enddo
   do j = 2, N-1
      do i = 2, N-1
         B(i,j) =(A(i+1,j)+A(i-1,j)+A(i,j+1)+A(i,j-1))*0.25
      enddo
   enddo
enddo
```

FIG. 1. Example of an iterative data parallel loop.

In previous work [1, 16], we developed a theoretical framework for analyzing near-neighbor communication in iterative data parallel loops. Since we use the same framework in this paper, we summarize relevant work. Given an iterative parallel loop that updates a single, two-dimensional, global array as in Fig. 1, the communication is determined by the *stencil*, S, which is the offsets of the array accesses:

$$S = \{(n_{h1}, n_{v1}), \ldots, (n_{hk}, n_{vk})\}.$$

For example, for Fig. 1,

$$S = \{(0, -1), (0, 1), (-1, 0), (1, 0)\}$$

Each offset pair in S is an *access vector*. Figure 2 shows the access vectors for a square partition under the stencil $\{(1, 0), (2, 0)\}$. A loop partitioning, ρ, is a mapping from the iteration space to a processor identification number (PID). The communication is defined to be the number of data points read (in each cycle) by a particular processor that are computed by another processor. Note that the first offset in each pair induces communication across the horizontal plane, as in Fig. 2, while the second offset induces communication across the vertical plane. Therefore, stencil elements provide communication per unit length across horizontal and vertical partition borders. For rectangular partitions of dimensions $h \times v$, the communication is expressed as $C_p = n_h h + n_v v$, where n_h and n_v are measures of communication along the horizontal and vertical dimensions. These are obtained from S by applying either the additive or the max–min construction procedure. The calculation of communication weight per orientation using the additive construction assumes that

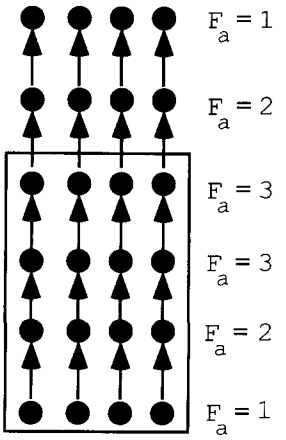

FIG. 2. The partition exterior is heavily referenced.

each reference made to a point outside the partition requires interprocessor communication, and is given by

$$n_h^a = \sum_{i=1}^{k} |n_{hi}|. \tag{3.1}$$

The calculation of communication weight using the max–min construction procedure assumes that interprocessor communication is only required once to establish a local copy of a datum and all successive reads (for the duration of the cycle) can be performed locally. In this case, the communication weight per orientation is given by $n_h = n_h^+ + n_h^-$, where $n_h^+ = \max(\{n_{hi} | n_{hi} \geq 0\} \cup \{0\})$ and $n_h^- = |\min(\{n_{hi} | n_{hi} < 0\} \cup \{0\})|$, where n_{hi} is the first element of the ith access vector. The constructions for n_v are analogous.

For multiprocessor systems with caches, the communication weights are converted from the number of data points shared between processors to the number of cache lines shared between processors [1]. Let l be the number of data points per cache line. Assuming column-major storage, the number of data points required per unit length along the horizontal boundary must be rounded up to the number of cache lines required per unit length $n_h^c = \lceil n_h/l \rceil$. For the vertical boundary we must compensate for the aggregation of multiple references into a single cache line, $n_v^c = n_v/l$. For example, if $l = 4$ and $n_h = n_v = 1$, $n_h^c = 1$, since one cache line is required per unit length along the horizontal border, and $n_v^c = 1/4 = 0.25$, since one of every four references along the vertical border requires a new cache line.

Assuming that $\mathcal{P}$, the number of processors, is fixed for all cycles, we prove in [1] that

$$h_{\text{opt}} = \sqrt{\frac{n_v N^2}{n_h \mathcal{P}}} \quad v_{\text{opt}} = \sqrt{\frac{n_h N^2}{n_v \mathcal{P}}}. \tag{3.2}$$

For a machine with private caches, n_h and n_v are replaced by n_h^c and n_v^c in the above. A more detailed treatment of this work is available elsewhere [1, 16].

In a private-cache bus-based multiprocessor, the movement of data between cache and main memory is managed by the hardware and the specification of the loop partition, ρ, determines the communication overhead [16]. In a scalable multiprocessor such as the BBN Butterfly, there is an additional degree of flexibility because the data assignment can be specified by the software and affects the communication time. A data assignment, σ, is a mapping from $\{1, 2, \ldots, N\} \times \{1, 2, \ldots, N\} \rightarrow \{1, 2, \ldots, \mathcal{P}\}$, i.e., a mapping from a data element to a processor, a. The element is stored in a memory that is closer to processor a than any other processor. In this paper, we determine a data assignment for a single global array based on the predetermined loop partition and the references made to the array in the code body. T_c, the communication time incurred by a processor on each cycle, is a function of the loop partition and the data assignment. There are two ways to reduce the impact of T_c on the execution time of the loop. First, T_c can be reduced by altering ρ and σ. Second, T_c can be overlapped with the computation time of the loop.

4. DATA ASSIGNMENTS

Three factors should be considered in specifying the data assignment. First, the consistency requirements may limit the possible locations for data. Second, subject to the first constraint, data that a processor references heavily should be assigned to the private levels of that processor. A third factor in determining data placement is the storage size available at each level in the hierarchy. Other researchers have developed efficient techniques for handling storage size limitations [13, 29] which can be applied following the use of our techniques.

In the general data assignment problem, an element of the global array may be assigned to several distinct storage locations in the multiprocessor memory hierarchy. First, we restrict the problem by simplifying the memory hierarchy to consist of two levels, i.e., local and remote memory. Each processor has fast access to an associated local memory and slower access to the remote memory, which is similar to a TC2000 when the private caches are not considered. Second, we require that each element of the global array be present in precisely one of the processors' local memories. These two restrictions simplify the data assignment problem to a data partitioning problem. We consider the optimization of the one-to-one mapping from each element of the global array to a particular local memory.

4.1. Data Partitioning

We assume that a loop partition ρ has been specified to minimize the data points exchanged between processors using ADP. We denote the rectangle that has its lower left-hand corner at coordinates (i_1, j_1) and its upper right-hand corner at coordinates (i_2, j_2) as $[i_1 : i_2, j_1 : j_2]$. Therefore, the partition of the iteration space assigned to processor a under loop partition ρ is the rectangle $[i_1 : i_2, j_1 : j_2]$, with lower left-hand corner (i_1, j_1) and upper right-hand corner (i_2, j_2). The *target set of y under S*, where $y = (i, j) \in A$, the global matrix, and S is the stencil set, is $T_S(y) = \{(i + n_{h1}, j + n_{v1}), \ldots, (i + n_{hk}, j + n_{hk})\}$ and gives all data points required for the computation of y [16]. The *read set*, $\mathcal{R}(a, \rho)$, is the set of data which are read by processor a under loop partition ρ, $\mathcal{R}(a, \rho) = \{x | \exists y \text{ s.t. } \rho(y) = a \text{ and } x \in T_S(y)\}$. The *write set*, $\mathcal{W}(a, \rho)$, consists of data which are written by processor a under loop partition ρ. Note that $\mathcal{W}(a, \rho) = \{(i, j) \in N \times N | \rho(i, j) = a\}$ since each iteration updates $A(i, j)$. The *use set*, $\mathcal{U}(a, \rho) = \mathcal{R}(a, \rho) \cup \mathcal{W}(a, \rho)$, consists of data that are read or written by processor a under the loop partition, ρ. The *use reference frequency* of a data point $A(i, j)$ by a processor a is the number of times processor a references $A(i, j)$, and is denoted $F_{(U,a)}(i, j)$. Consider a data partition σ that maps each element of the global array to a single processor. The *local references* of processor a are $L(a) = \{(i, j) \in N \times N | \sigma(i, j) = a\}$.

For this section we assume a true partition of the data set, i.e., a division of the data set into mutually disjoint subsets. If t_l is the latency to access local memory and t_r is the latency to access remote memory, the latency for accesses to locations in $L(a)$ is t_l, while the latency for accesses to all other locations is t_r. Then, T_c, the communication time, reduces to

$$T_c = \sum_{(i,j) \in L(a)} (F_{(U,a)}(i, j) t_l) + \sum_{(i,j) \notin L(a)} (F_{(U,a)}(i, j) t_r). \quad (4.1)$$

The objective of minimizing execution time is equivalent to the objective of minimizing communication time, since the loop partition evenly distributes a fixed amount of computational work among the processors. The fact that the loop partition is fixed also implies that $\sum_{(i,j) \in N \times N} F_{(U,a)}(i, j)$ is constant. Therefore, the performance is optimized when $\sum_{(i,j) \in L(a)} F_{(U,a)}(i, j)$ is maximized.

Consider Fig. 2, which shows the target sets for a square partition under the stencil $\{(1, 0), (2, 0)\}$. The values of $F_{(U,a)}$ are given for various rows. Observe that the row across the top border of the square has a frequency access of 2, while the row along the bottom border has a frequency access of 1. Therefore, the data partition that maximizes local accesses should hold the row across the top border rather than the row along the bottom border. The data partition that maximizes local accesses is obtained by shifting the loop part $\{(i, j) | \rho(i, j) = a\}$ up by one row.

In general, the information provided by the stencil can determine how to shift the data partition. By restricting the discussion to rectangles, we can consider shifts along two dimensions: vertical shifts and horizontal shifts. We will discuss vertical shifts, the horizontal shifts being analogous. Consider a stencil $S = \{(n_{h1}, n_{v1}), \ldots, (n_{hk}, n_{vk})\}$ that is sorted by n_{hi}, i.e., $n_{h1} \le n_{h2} \le \ldots \le n_{hk}$.

THEOREM 1. *Let* $r_n = |\{n_{hi} \text{ s.t. } n_{hi} < 0\}|$ *(where the bar notation refers to set cardinality),* $r_z = |\{n_{hi} \text{ s.t. } n_{hi} = 0\}|$, *and* $r_p = |\{n_{hi} \text{ s.t. } n_{hi} > 0\}|$. *Upward shifts are made when* $r_p > r_n + r_z$, *and downward shifts are made when* $r_n > r_p + r_z$. *Assuming* $|i_1 - i_2| \gg max\{|n_{h1}|, |n_{hk}|\}$, *the shift, s, from the loop part that maximizes the number of local references made by a processor,* $max_s(\sum_{i=i_1+s}^{i_2+s} F_{(U,a)}(i))$, *is*

$$s_{opt} = \begin{cases} n_{hi} & \text{where } i = \dfrac{k+1}{2},\ k \text{ odd.} \\ n_{hi} & \text{where } i = \dfrac{k}{2},\ k \text{ even.} \end{cases} \quad (4.2)$$

Proof. r_n and r_p represent the quantity of references made across the i_1 and i_2 borders, respectively. r_z represents the volume of references made exclusively to the partition interior. Shifting is done in order to include data that are more heavily referenced than some data which are currently included. Vertical shifts can be done either in the upward or downward direction. An upward shift includes data higher than i_2 at the cost of excluding data near i_1. Therefore, upward shifts should be done when $r_p > r_z + r_n$. Similarly, downward shifts should be done when $r_n > r_z + r_p$.

We focus on an upward shift, the downward shift being analogous. The number of local accesses is $\max_s(\sum_{i=i_1}^{i_2} F_{(U,a)}(i) + \sum_{i=i_2+1}^{i_2+s} F_{(U,a)}(i) - \sum_{i=i_1+1}^{i_1+s} F_{(U,a)}(i-1))$. Since $\sum_{i=i_1}^{i_2} F_{(U,a)}(i)$ is constant with respect to s, the objective, this is restated as $\max_s[\sum_{l=1}^{s}(F_{(U,a)}(i_2 + l) - F_{(U,a)}(i_1 - 1 + l))]$. Assuming $|i_1 - i_2| \gg \max\{|n_{h1}|, |n_{hk}|\}$, we have $F_{(U,a)}(i_2 + l) = |\{n_{hi} \text{ s.t. } n_{hi} \ge l\}|$. Therefore, $F_{(U,a)}(i_2 + l)$ is a monotonically decreasing function of l decreasing from r_p for $l = 1$ to 0 for $l > n_{hk}$. Similarly, $F_{(U,a)}(i_1 - 1 + l) = |\{n_{hi} \text{ s.t. } n_{hi} < l\}|$ is a monotonically increasing function of l. Recall that $|i_1 - i_2| \gg \max\{|n_{h1}|, |n_{hk}|\}$, $(F_{(U,a)}(i_2 + l) - F_{(U,a)}(i_1 + l))$ is a monotonically decreasing function of l decreasing from $r_p - (r_n + r_z)$ for $l = 1$ to $-(r_n + r_z + r_p)$ for $l > n_{hk}$. Therefore, the summation is maximized when $F_{(U,a)}(i_2 + s_{opt}) - F_{(U,a)}(i_1 - 1 + s_{opt}) \ge 0$ and $F_{(U,a)}(i_2 + s_{opt} + 1) - F_{(U,a)}(i_1 - 1 +$

$s_{opt} + 1) < 0$, i.e., when $|\{n_{hi}$ s.t. $n_{hi} \geq s_{opt}\}| \geq |\{n_{hi}$ s.t. $n_{hi} < s_{opt}\}|$ and $|\{n_{hi}$ s.t. $n_{hi} \geq s_{opt} + 1\}| < |\{n_{hi}$ s.t. $n_{hi} < s_{opt} + 1\}|$. Since s_{opt} partitions the n_{hi} set into two subsets, the optimum shift, s_{opt}, is given by Eq. (4.2). And so the claim is shown. ■

The above result can be extended as follows. Under the assumption that the loop part dimensions are much larger than the stencil, the optimal data partition within a constant factor involving the product of $\max(|n_{hi}|)$ and $\max(|n_{vi}|)$ is obtained by applying the optimal shifts in each direction as specified by Theorem 1. The proof is not included due to space limitations. The main result of this section is that for a multiprocessor memory hierarchy consisting of just local and remote memories, a simple procedure derived from Theorem 1 can be used to find the optimal data partition that minimizes communication time for iterative parallel loops.

4.2. Hardware Redundancy

In this section, we expand the scope of the data assignment problem to include multiprocessor memory hierarchies with private caches. The *private level* of a memory hierarchy is only accessible to a particular processor, e.g., the private caches on the TC2000. In such a system, frequently accessed data can be copied into a private cache, thus introducing redundancy in storage, i.e., multiple copies of a datum. This redundancy is referred to as *hardware redundancy* because it is largely managed by the hardware. Only one logical address is used for all the multiple copes of the datum. In contrast, *software redundancy*, which is discussed in Section 7, involves multiply addressed copies of the same datum in different local memories managed explicitly in software.

In contrast to the data partitioning problem, where the partition only influenced the performance, hardware redundancy also involves correctness and consistency considerations. In the absence of hardware or software coherency schemes, only those data elements that are used exclusively by a processor can be cached by that processor. In this section, we assume that the data have been partitioned as described previously into the different local memories. We exploit the lower latency of the cache by selectively declaring regions of the data part assigned to a local memory to be cacheable.

Every array location has read and write characteristics (with respect to a given processor) in the set {*shared*, *exclusive*, *no*}. For example, an array location which is exclusively read from and written to by a single processor is an exclusive read–exclusive write location with respect to that processor. Potentially, locations could be classified into as many as nine different categories of read–write characteristics. However, only four of these categories are of interest in our current context. Since the single assignment property has been assumed, all categories with the shared-write characteristic are eliminated. Cycle-by-cycle communication is only induced (assuming sufficient local storage) by array locations which are both read from and written to in a single cycle. Read-only arrays have fixed values across all cycles, and represent startup communication which should not influence partitioning decisions.

Global array locations fall into one of four sets. A datum $A(i, j)$ belongs to the exclusive read–write set of processor a if $A(i, j) \in \mathcal{R}(a, \rho) \cap \mathcal{W}(a, \rho)$ and $A(i, j) \notin \mathcal{R}(p, \rho) \cup \mathcal{W}(p, \rho)\ \forall p \neq a$. A datum $A(i, j)$ belongs to the shared read–exclusive write set of processor a if $A(i, j) \in \mathcal{W}(a, \rho)$ and $A(i, j) \notin \mathcal{W}(p, \rho)\ \forall p \neq a$ and $\exists p \neq a$ s.t. $A(i, j) \in \mathcal{R}(p, \rho)$. A datum $A(i, j)$ belongs to the shared read–no write set of processor a if $A(i, j) \in \mathcal{R}(a, \rho)$ and $A(i, j) \notin \mathcal{W}(a, \rho)$ and $\exists p \neq a$ s.t. $A(i, j) \in \mathcal{W}(p, \rho)$. In addition, the data not used by processor a belong to the no read–write set (NRW) of processor a. An important feature of our scheme is that these sets are identified at compile-time as a function of the array dimensions, the number of processors, and the communication parameters $(n_h^+, n_h^-, n_v^+, n_v^-)$.

THEOREM 2. *Given a loop part of processor a, $\rho(a) = [i_1 : i_2, j_1 : j_2]$ of dimensions $h \times v$, the ERW set is $[i_1 + n_v^+ : i_2 - n_v^-, j_1 + n_h^+ : j_2 - n_h^-]$, the SREW set is $\rho(a) \backslash \text{ERW} = [i_1 : i_2, j_1 : j_2] \backslash [i_1 + n_h^+ : i_2 - n_h^-, j_1 + n_v^+ : j_2 - n_v^-]$, the SRNW set is contained by $[i_1 - n_h^- : i_2 + n_h^+, j_1 - n_v^- : j_2 + n_v^+] \backslash [i_1 : i_2, j_1 : j_2]$, and the NRW set is a superset of $N \times N \backslash (\text{ERW} \cup \text{SRNW} \cup \text{SREW})$.*

Proof. From the code construct, observe that every iteration writes only one data element. Therefore, the write set, $\mathcal{W}(a, \rho) = \rho(a) = [i_1 : i_2, j_1 : j_2]$, is the exclusive write set. A subset of the exclusive write set is the shared read–exclusive write, which is given by

$$\{(i, j) \in [i_1 : i_2, j_1 : j_2] \mid \exists (\iota, \kappa) \notin [i_1 : i_2, j_1 : j_2] \text{ s.t. } (i, j) \in T_S(\iota, \kappa)\}.$$

(ι, κ) fits at least one of the following criteria: $\iota < i_1$, or $\iota > i_2$, or $\kappa < j_1$, or $\kappa > j_2$. Therefore, the shared read–exclusive write set is

$$\begin{aligned}
&\{(i, j) \in [i_1 : i_2, j_1 : j_2] \mid \exists (\iota, \kappa) \text{ with } \iota < i_1 \text{ or } \iota > i_2 \text{ or } \kappa < j_1 \text{ or } \kappa > j_2 \text{ s.t. } (i, j) \in T_S(\iota, \kappa)\} \\
&\quad = \{(i, j) \in [i_1 : i_2, j_1 : j_2] \mid i < i_1 + n_h^+ \text{ or } i > i_2 - n_h^- \text{ or } j < j_1 + n_v^+ \text{ or } j > j_2 - n_v^-\} \\
&\quad = [i_1 : i_2, j_1 : j_2] \backslash [i_1 + n_h^+ : i_2 - n_h^-, j_1 + n_v^+ : j_2 - n_v^-].
\end{aligned}$$

The above expressions of the SREW set and the EW set yield the expression for the ERW set. The shared read–no write set is

$$\{(i, j) \notin [i_1 : i_2, j_1 : j_2] \mid \exists (\iota, \kappa) \in [i_1 : i_2, j_1 : j_2] \text{ s.t. } (i, j) \in T_S(\iota, \kappa)\}$$

which, from the definition of the communication parameters, implies that at least one of the following is true: $i \leq i_2 + n_h^+$, or $i \leq i_1 - n_h^-$, or $j \leq j_2 + n_v^+$, or $j \geq j_1 - n_h^-$. And so the claim is shown. ■

For example, consider the stencil $S = \{(3, 2), (1, 3), (-1, -3), (-2, 3)\}$. Observe that $n_h^+ = 3$, $n_h^- = 2$, $n_v^+ = 3$, $n_v^- = 3$. Consider Fig. 3, where $\rho(a) = [i_1 : i_2, j_1 : j_2]$ and the ERW, SREW, and SRNW sets are shown. The empty corners of the outermost rectangle correspond to regions of the NRW set which are included in our approximation of the SRNW set as demonstrated in Theorem 2.

Our compile-time analysis of communication and the subsequent partitioning of the array into four sets with respect to each processor permits the introduction of caching schemes. The simpler scheme only caches the ERW set and does not require cache invalidates. The more sophisticated scheme achieves even smaller communication time by caching the entire exclusive write set and using cache invalidates to maintain consistency, i.e., by flushing the SREW from the cache at the end of each cycle, thus updating the copy of the SREW set in main memory.

Let us first analyze the simpler scheme without cache invalidates. In this scheme, at the beginning of the parallel section of the program, an array of the appropriate dimension is allocated by each processor in its local memory and declared cacheable. Each processor copies the portion of the array corresponding to its ERW set into this local array. Also, each processor allocates a noncacheable array for storing its SREW in its local memory. Subsequent references during the execution of the parallel section are made to the local copies. The communication time is reduced to $T_c = \sum_{(i,j)\in \text{ERW}} F_{(U,a)}(i, j)t_c + \sum_{(i,j)\in \text{SREW}} F_{(U,a)}(i, j)t_l + \sum_{(i,j)\in \text{SRNW}} F_{(U,a)}(i, j)t_r$. If cache invalidates are used, the communication time is expressed as $T_c = \sum_{(i,j)\in \text{ERW} \cup \text{SREW}} F_{(U,a)}(i, j)t_c + \sum_{(i,j)\in \text{SRNW}} F_{(U,a)}(i, j)t_r$, ignoring the time required for cache invalidates.

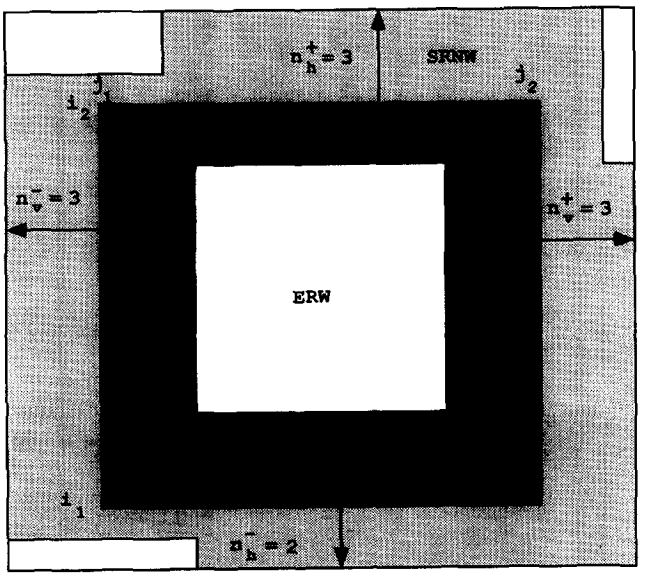

FIG. 3. The regions of a data set.

Consider the example presented in Fig. 1. Assuming a square partitioning with parts of size 25×25, and using the times supplied by BBN, i.e., $t_r = 1.89$ μs, $t_l = 0.60$ μs, and $t_c = 0.15$ μs, Eq. (4.1) yields $T_c = 571.56$ μs per cycle, the former expression yields $T_c = 331.51$ μs per cycle, and the latter yields $T_c = 290.31$ μs per cycle. Clearly, the ability to exploit an efficient loop partitioning strategy by moving large quantities of data high into the memory hierarchy has a significant impact on communication time.

5. EXPLOITING OVERLAP

Research on iterative parallel loops has focused on reducing communication time through exploitation of the memory hierarchy. An alternative approach to reducing the overhead of communication is to overlap computation and communication time. However, aspects of a processor's architecture can limit the maximum achievable overlap. For instance, the number of available registers may be too few to hold the partial results of many loop iterations, or the processor may lack special hardware required for a large maximum overlap (e.g., separate ports to local and global memory). In practice, programming and compilation techniques also influence the overlap achieved. Our objective is to focus on both reducing communication time and increasing overlap to reduce communication overhead.

Results by Callahan *et al.* [5] and Mangione-Smith *et al.* [21] are useful in the subsequent discussion. A processor's resources are broadly classified into compute and access resources [5]. The maximum performance of a particular loop is achieved once one of the resources is fully utilized. Accordingly, loops are classified into *compute-bound* and *memory-bound* loops. In our framework, loop iterations can be similarly classified as compute-bound if all data references can be satisfied by the cache or communication-bound if some data references require local or remote memory accesses.

The compute-bound set is the subset of $\rho(a) = [i_1 : i_2, j_1 : j_2]$ whose use set is contained by the ERW set of processor a, and is given by $\Xi = [i_1 + n_h : i_2 - n_h, j_1 + n_v : j_2 - n_v]$. Ξ is much larger than the rest of the itera-

tions to be performed, $\Gamma = \rho(a) \backslash \Xi$. We propose a compiler which schedules code for a communication-bound iteration from Γ together with a sufficient number of compute-bound iterations from Ξ. We refer to such a group of iterations as a *node*. Code scheduling within a node orders the instructions as follows: loads for the iterations from Ξ (which are all satisfied by the cache), loads for the iterations from Γ (which may require long latencies), execution of the iterations from Ξ (which are executed simultaneously with the loads of Γ), and execution of the iterations from Γ. Finally, the results computed by the iterations in the node are stored.

Despite a lack of hardware support, the overlapping of computation and communication can be accomplished on the BBN TC2000 through special code scheduling. The Motorola 88000 [22] issues one instruction on every cycle, unless there is a stall in the instruction issue unit. The instruction issue unit stalls when it must dispatch to a pipe which is full. The memory access pipe on the 88000 has three stages. The pending accesses to local and remote memory wait at the third stage for their data. Should the load pipe be filled with two memory operations that are waiting behind a pending memory operation, the instruction issue unit stalls on another memory access instruction. The feature of the 88000 that influences our methods for reducing communication overhead is the overlap possible between the access and floating point units that enables us to execute floating-point operations while waiting for memory.

Some architectural specifics of the Motorola 88000 point to fundamental limits to maximum overlap. The memory system of the 88000 operates in a pipelined fashion. The in-order operation of memory requires that all loads for a computation to be overlapped with a remote memory access must be issued before the remote access is issued. In order to avoid waiting on the remote memory access, data for compute-bound iterations must be in registers before initiating the remote load. The number of CPU registers seriously limits the maximum number of compute-bound iterations that can be overlapped, as the following analysis indicates.

Assume that a node consists of one communication-bound iteration requiring a single remote load and several compute-bound iterations. A remote load requires at least 38 cycles to complete. This latency is completely overlapped only if 38 floating point operations whose operands are already in registers can be issued. Assuming four floating point operations and four operands per compute-bound iteration for the code in Fig. 2, at least 38 registers are required. Composing a node using a square compute-bound tile of size 3×3 with each communication-bound iteration will reduce register requirements to approximately 25, but this introduces additional compilation complexity [4].

6. SOFTWARE REDUNDANCY

In software redundancy, the data assignment is extended to create additional copies of the data in the local memories of individual processors. Consistency is maintained by inserting separate stores in the instruction stream for each update. Software redundancy is a natural extension to the hardware redundancy already exploited to reduce latencies; e.g., one datum may be simultaneously in a memory location and in cache. In this section, we permit the data assignment to be a one-to-many mapping from the elements of the global array to processors. A particular element may appear in several local memories. We concern ourselves with *maximum software redundancy*, which is the replication of data elements so that each processor has a local copy of all elements in its use set. The corresponding data assignment, σ, maps (i, j) to those processors that access (i, j), and is the inverse of the use set mapping, $\mathcal{U}(a, \rho)$: $\sigma(i, j) = \{a | (i, j) \in \mathcal{U}(a, \rho)\}$.

Two factors to be considered in using software redundancy are the extra memory space and additional consistency updates it incurs. In the following, we quantify each of these factors. The amount of memory space allocated per processor is $|\mathcal{U}(a, \rho)|$ as compared to $|L(a)|$ previously. Note that $|\mathcal{U}(a, \rho)| = |[i_1 - n_h^- : i_2 + n_h^+, j_1 - n_v^- : j_2 + n_v^+]| = (v + n_h)(h + n_v)$, while $|L(a)| = hv$. Therefore, the additional storage required is $|\mathcal{U}(a, \rho)| - |L(a)| = vn_v + hn_h + n_h n_v$ and the fractional increase in memory requirements is

$$\frac{vn_v + hn_h + n_h n_v}{hv} = \frac{n_v}{h} + \frac{n_h}{v} + \frac{n_h n_v}{hv}. \tag{6.1}$$

Observe that, since h and v are typically at least an order of magnitude larger than n_h and n_v, the value computed by Eq. (6.1) is small. Since our analysis accurately identifies elements used by a particular processor, even maximum software redundancy only marginally increases storage requirements.

Let us examine the impact of maximum software redundancy on consistency traffic.

LEMMA 1. *When using maximum software redundancy, the increase in the number of local loads is* $hn_h^a + vn_v^a$, *where* n_h^a, n_v^a *are quantified by Eq.* (3.1) *and the increase in the number of remote stores is* $hn_h + vn_v$ *and the savings in communication time is* $(hn_h^a + vn_v^a)(t_r - t_l) - (hn_h + vn_v)t_r$.

Proof. The increase in the number of local loads is equal to the number of read references made by a processor to its SRNW set. The communication parameters obtained by the additive construction procedure quantifies

the number of such references per unit length of the part border [1, 16]. Therefore, the increase in the number of local loads is $hn_h^a + vn_v^a$. There is a corresponding decrease in the number of remote loads, resulting in a net decrease in communication time of $(hn_h^a + vn_v^a)(t_r - t_l)$.

A data element in the SRNW set is computed by another processor. When the maximum software redundancy data assignment scheme is used, such points require consistency updates by another processor. Therefore, the number of points in the SRNW set is equal to the number of extra stores required to implement this scheme. The size of the SRNW set is approximately $hn_h + vn_v$. Therefore, the communication time is increased by $(hn_h + vn_v)t_r$. ■

For data which are written by one processor and read by another, the consistency is maintained on the TC2000 among the multiple copies by the processor performing the update. The traditional method of transferring data between processors is a demand-based, or *pull*, protocol. Software redundancy replaces this with a *push* protocol. A processor which is writing a datum back to memory has a list of storage locations which also must be updated. The processor updates its copy of the datum, along with all copies on the list. The net effect is that the processor "pushes" the new data value into the local memory of the processor which is waiting on the datum.

7. ADAPT

The ADAPT (automatic data allocation and partitioning tool) system generates code to automatically manage data assignments for iterative parallel loops. ADAPT consists of a set of routines which are implemented within the PAT (Parallelizing Assistant Tool) system developed at Georgia Tech by Appelbe *et al.* [27]. Existing facilities within PAT were used to identify triply nested loops in sequential FORTRAN code and to analyze the code body of the innermost loop for array references. The ADAPT routines analyze these references to determine the optimal aspect ratio. In addition, BBN parallel FORTRAN is generated to implement a partitioning based on the size of the iteration space, the number of processors, and the aspect ratio.

In addition to partitioning, ADAPT also exploits the memory hierarchy of the BBN by declaring shared arrays to be cacheable. ADAPT uses the access vectors obtained from an analysis of the input code to determine the SREW and SRNW sets for each part. Cache flush instructions are then inserted into the code to flush the SREW and SRNW sets after each cycle. The false sharing of data between processors introduced by cache lines further complicates consistency maintenance and will be discussed shortly.

7.1. ADAPT's Preamble and Run-Time Partitioning

Assuming the stencil set is fixed, partitioning at compile time is desirable in order to avoid the expense of communication analysis at run time. However, a strictly static partitioning approach is not practical for many programs where the array and loop bounds, as well as the number of processors, are not known at compile time. ADAPT's solution to this dilemma lies in the recognition of the partitioning problem as two distinct phases: communication analysis (i.e., determination of the optimal aspect ratio from the access vectors) and partition generation (i.e., the determination of the part boundaries for each processor executing the parallel loops). At compile time, ADAPT performs communication analysis and collects other information from the program which is required for partition generation, i.e., the bounds of the parallel loops (which may be expressions that cannot be evaluated at compile time). This information is placed within a preamble which is inserted just in front of the parallel loop in the output code generated by ADAPT. At run time, each processor executes the preamble prior to the first cycle, thus completing partition generation.

Eq. (3.2) give the dimensions of a rectangular partition with parts of a given size (i.e., $N^2/\mathcal{P}$) that has the minimum value of $C_p = hn_h + vn_v$. However, the ability of the partition to tessellate the iteration space is not guaranteed. The partition generation algorithm of the ADAPT preamble takes a different approach: it considers the set of rectangular partitions that tessellate the iteration space, and selects the one with the aspect ratio that is closest to the optimal aspect ratio.

Assume an $N \times M$ iteration space. For a given number of processors, $\mathcal{P}$, the set of rectangular partitions which tessellate can be generated from the set of divisors of $\mathcal{P}$. Let $qr = \mathcal{P}$, and assume (for the moment) that q divides N and r divides M. In a treatment similar to the OPTAL algorithm of Polychronopoulos [24], the rows of the iteration space are assigned into q classes and the columns are assigned into r classes. The part to be executed by processor p is located in row class (p/r) and in column class (p and r). For such a partitioning, the aspect ratio is Nq/Mr. The preamble of ADAPT examines this aspect ratio for all possible values of q and r. The values of q and r for which the aspect ratio is closest to the optimal aspect ratio, i.e., h_{opt}/v_{opt} is chosen as the partition.

Now consider the case when q does not divide N evenly, i.e., let $N \bmod q = 0_q$, where $0_q \neq 0$. In such a case, load imbalance is introduced. The first o_q row classes contain an extra row while the remaining $q - 0_q$ row classes contain $[N/q]$ rows. The case when r does not divide M evenly is handled analogously. Under these assumptions the maximum number of iterations that any processor must execute in addition to the original $[N/q]$

$[M/r]$ iterations is $[N/q] + [M/r] - 1$. By replacing the integer-valued functions with real-valued functions, the maximum relative load imbalance is

$$\left(\frac{N}{q} + \frac{M}{r} - 1\right) \Big/ \left(\frac{N}{q}\frac{M}{r}\right)$$

which simplifies to $(Nr + Mq - qr)/(NM)$. And, since values of N and M are typically orders of magnitude greater than q and r, the relative load imbalance introduced by ADAPT's partitioning scheme is usually negligible.

7.2. *False Sharing and Consistency*

In order to exploit the memory hierarchy of the BBN TC2000, ADAPT declares the global arrays used within the iterative parallel loops to be cacheable. Since the cache is a private level of the TC2000 memory hierarchy, automatic consistency of data is not provided. ADAPT maintains the consistency of data in software through the use of cache flush instructions. ADAPT estimates the SREW and SRNW sets for each partition using the access vectors obtained from the input code and Theorem 2. ADAPT then generates cache flush instructions to flush the SREW and SRNW sets of each part. These instructions are inserted after the code responsible for updating matrices and are synchronized using a barrier, so the activities enforced by ADAPT within a single cycle are: (1) Update partition elements, reading most recently updated copies of shared data elements from memory. (2) Flush shared data elements into global memory. (3) Synchronize at a barrier to prevent processors from beginning the next cycle before the shared data is resident in global memory.

Assuming column major storage, contiguous data points within a column are located in contiguous memory locations. For the Motorola 88000 processors used in the BBN TC2000, the cache line size is 16 bytes and the floating point data type is 4 bytes long. Therefore, assuming the matrix begins on a cache line boundary, exactly four data elements from the global matrix are contained on a single cache line. This inclusion of four data points on a single cache line complicates the consistency maintenance on the BBN TC2000. For example, consider a cache line, the first of whose points is updated by a particular processor while the remaining three points are updated by another processor. During the update of their respective partitions, the two processors read data elements from the locations contained on the cache line and a copy of the cache line is created in each processor's cache. The first processor updates the first element on the cache line, and now possesses a line containing one current value and three "stale" values. Similarly, the second processor possesses a cache line containing one stale value and three current values. After the partition updates, the cache lines are flushed back into global memory. Assume that the first processor flushes the cache line, followed by the second processor. The flush performed by the second processor replaces the value in memory that was updated by the first processor with the stale value possessed by the second processor. Indeed, regardless of the order in which the two processors flush their cache lines, stale data will reside in memory.

Though the partitioned loop does not contain dependencies between array values within a cycle, there are output dependencies that must be maintained on cache lines for correctness. The key to a general and elegant solution to this problem lies in recognizing the need to maintain certain output dependencies on cache lines. Dependencies are usually maintained by inserting synchronization operations. If in a particular parallel segment a cache line is updated by at most s processors, correct execution is in general obtainable by inserting s synchronization operations so that in each phase no more than one processor updates any cache line. Following each phase, all processors flush all cache lines that can possibly be updated in that phase. All processors except at most one have a clean copy of the line and do not write to memory. Only the processor having the dirty copy of the line writes to memory.

In the context of rectangular partitioning of iterative parallel loops, the false sharing problem has to be addressed for those cache lines in the horizonal borders of each part. All other cache lines are exclusively updated by one processor. The horizontal border cache lines may be updated by two processors. Therefore, we further subdivide each parallel segment into two segments and insert an additional synchronization barrier as follows: (1) Update the top half of each part, reading the most recently updated copies of shared data elements from memory. (2) Flush shared data associated with the top half of each part into global memory. (3) Synchronize at a barrier to prevent simultaneous access of shared cache lines by processors. (4) Update the bottom half of each part, reading the most recently updated copies of shared data elements from memory. (5) Flush shared data associated with the bottom half of the part into global memory. (6) Synchronize at a barrier to prevent processors from beginning the next cycle before the shared data are resident in global memory.

8. EXPERIMENTAL RESULTS

Experiments were run on a 45-processor BBN TC2000 at Argonne National Laboratories. For the first suite of experiments, the code presented in Fig. 1 was restruc-

tured by hand to implement various data assignments and varying degrees of overlap. In the second suite of experiments, the SHOPF code segment from the BBN manual was restructured using the ADAPT system.

8.1. Data Assignment Experiments

For the first set of experiments in this subsection, we used a simple column partition to simplify the implementation of various data assignments and overlap. To further simplify the problem, each processor communicated across only one boundary. The relative performance of the candidates in this experiment is unaffected by these choices.

The data assignments for the sample code were made by considering the placement of the ERW and SRNW sets of each processor. The following notation is used in this section to abbreviate the levels of the hierarchy; "c" stands for cache, "lm" for local memory, and "rm" for remote memory. The data assignment is specified by an ordered pair, (hierarchy level holding the SRNW, hierarchy level holding the ERW). The SREW set is automatically placed with the SRNW set. The location of both sets is referred to as the SRNW location for brevity. Execution times from the BBN TC2000 in μs are given for various data assignments in Table I.

The largest execution time occurs, not surprisingly, when no special attention is paid to the data assignment. This is the case labeled (rm, rm) in Table I, and corresponds to scattering the array among the memory modules executing the program, using the BBN "scatter" command. In order to make a fair comparison, the time obtained for the "scatter" data distribution is compared to a data assignment with no redundant storage. Our data partitioning scheme described in Section 5.1 places each processor's exclusive write set in its local memory, and is the (rm, lm) case in Table I. The observed decrease in execution time is a factor of 3.63, for a savings of 72.45 percent. For this application, utilization of a data assignment dramatically improves the performance of the BBN.

The experiments are based on the placement of the ERW set and the SRNW set for each processor. In our experiments, software redundancy and hardware redundancy are both exploited. The (rm, c) case results from

TABLE I
Execution Times in μs for Various Data Assignments

SRNW \ ERW	rm	lm	c
rm	297298	81710	59750
lm	—	76017	54054

TABLE II
Relative Performance of a Processor for Various Data Assignments

SRNW \ ERW	lm	c
rm	1.00	0.73
lm	0.93	0.66

the implementation of hardware redundancy using local arrays which are declared to be cacheable. The local arrays are used in this section to illustrate the performance found at various levels of the memory hierarchy and in the implementation of overlap. Software redundancy, as described in Section 7.1, is used when the shared points are placed in the local memory. The results presented in Table II demonstrate the obvious observation that reducing the latencies for all data points results in the minimum execution time. However, we can separately analyze the effects of software redundancy and hardware redundancy for this application. The experimentally observed reduction in execution time using hardware redundancy in 27%. The observed reduction in execution time using software redundancy is 7%.

In order to test the effects of data assignment on a more complex loop partition, the code in Fig. 1 was partitioned using squares and executed on 16 processors of the BBN TC2000. An analysis of the loop in Fig. 1 when partitioned for $N = 100$ and $\mathcal{P} = 16$ indicates that 96% of all references are made to the ERW set, while 4% of all references are made to shared points. When using square partitions, the number of accesses made to the ERW set is proportional to the area of the partition, which grows quadratically in N. Meanwhile, the number of accesses made to the shared points is proportional to the perimeter of the partition, which grows linearly in N. Therefore, as N grows, the performance of the (rm, c) assignment should improve relative to the (rm, rm) assignment. Experiments varying N from 100 to 400 were run on the BBN TC2000. The ratio of the execution time of the (rm, rm) to the execution time of the (rm, c) assignment obtained is presented in Fig. 4. Note that, as N increases from 100 to 240, the ratio increases. However, as N exceeds 240, the ratio drops dramatically. This is because the Motorola 88000 processors have only 16 kilobyte of data cache, and $N = 240$ is the largest value of N for which the ERW set entirely fits in the cache.

In our experiments, the node construct overlapped one load of a shared point (in local or remote memory) with a number of iterations from the compute bound set. Due to the limited number of registers, any attempted overlap of more than three iterations resulted in partially computed

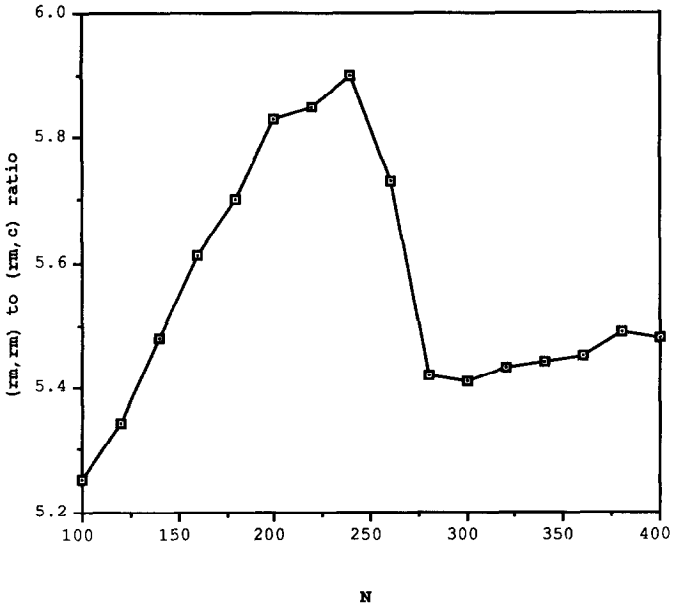

FIG. 4. The ratio of (rm, rm) to (rm, c) as N increases.

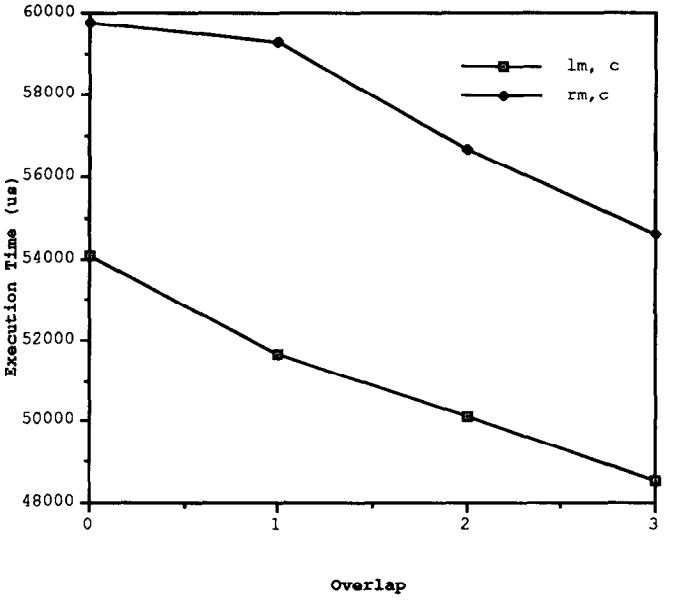

FIG. 5. Execution time in μs vs degree of overlap. (⊡) lm, c; (◆) rm, c.

results being spilled back to memory. The execution times (in microseconds) are given for the (lm, c) data assignment and the (rm, c) data assignment in Fig. 5 as the number of overlapped iterations increases. For an overlap of three, the improvement in execution time is 8.65% for the (rm, c) case and 10.21% for the (lm, c) case.

8.2. ADAPT

The previous experiments have compared one data assignment with another, using the same partitioning method for each program. It is important to compare the result of our partitioning with more traditional methods of scheduling parallel programs. The SHOPF routine [3] is a code segment extracted from a fluid flow application. It involves the update of a global matrix using the stencil {(2, 0), (1, 1), (1, 0), (1, −1), (0, 2), (0, 1), (0, 0), (0, −1), (0, −2), (−1, 1), (−1, 0), (−1, −1), (−2, 0)}. Two versions of the code were used in experiments. The first version used chunk scheduling. The second version was restructured code generated by ADAPT.

Three experiments were conducted on the code generated by ADAPT. ADAPT was used to generate code with varying aspect ratios to determine the impact of altering the aspect ratio on performance. The possible aspect ratios for 16 processors, along with their column and row assignments, are given in Table III. Using the max–min construction, $n_h = 4$ and $n_v = 4$, so the optimal aspect ratio is 1. However, since a cache line in the BBN TC2000 system contains more than one data point, the effects of cache lines on communication must be considered as detailed in [1]. From this treatment, the optimal aspect ratio is 0.25, as is demonstrated experimentally in Table III. Facilities within ADAPT to compensate for cache line effects have been added. In order to observe the effects of matrix size on performance, both versions of the code were run on 16 processors for varying matrix sizes. The exploitation of the BBN memory hierarchy improved performance of the ADAPT-generated code relative to the code using chunk scheduling from a factor

TABLE III
Execution Time of SHOPF with $N = 200$, $\mathcal{P} = 16$

Aspect ratio	Processors dividing columns	Processors dividing rows	Time (sec.)
0.0625	16	1	32.12
0.25	8	2	28.51
1	4	4	29.18
4	2	8	32.53
16	1	16	45.05

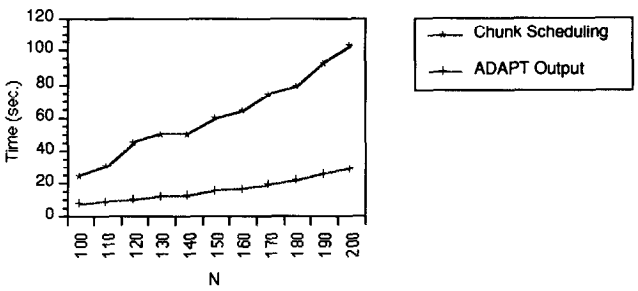

FIG. 6. Execution time of SHOPF with $\mathcal{P} = 16$, aspect ratio of 0.25.

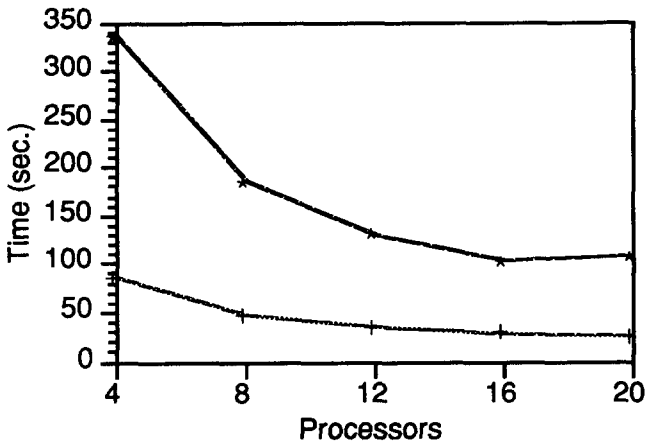

FIG. 7. Execution time of SHOPF with $N = 200$, aspect ratio of 0.25. (☆) Chunk scheduling, (+) ADAPT output.

of 3.27 when $N = 100$ to 3.69 when $N = 200$, as shown in Fig. 6. Finally, the code was compared for various numbers of processors in Fig. 7.

9. CONCLUSION

The nonuniformity of memory access times found on large-scale, shared-memory multiprocessors is a direct result of scaling the systems to large numbers of processors. However, this is not an unfortunate result that must be hidden from the compiler. If exposed, the compiler can exploit the nonuniformity to extract even greater performance. In this paper, we examined automatic methods for reducing the impact of communication time on the execution time of a parallel loop. We identify and focus on iterative data parallel loops. For the optimal loop partitions generated by ADP, we develop an optimal data partition that minimizes communication time.

The ADAPT (automatic data allocation and partitioning tool) system was developed in order to automatically partition programs. The use of ADAPT on a fluid flow code segment improved performance by over a factor of 3 over the partitioning method suggested by BBN. In order to reduce communication overhead even further, we consider the overlapping of compute-bound iterations with memory-bound iterations. Certain machine features, e.g., the size of the register file and the single memory access pipe on the Motorola 88000, limit the maximum achievable overlap.

Many opportunities exist for future work in this area. We are currently working on extending this analysis to multiple loops with potentially different access patterns. A classification of loops to determine the optimal amount of software redundancy may lead to improved performance. Improvement of maximum overlap can be achieved through a two-level loop blocking scheme which provides for maximum reuse of registers. Also, our analysis can be extended to other memory hierarchies. Additionally, our approach must be generalized to a wider range of numerical applications. These are major areas for future work.

ACKNOWLEDGMENTS

The authors thank Bill Appelbe, Kurt Stirewalt, and particularly Kevin Smith for their assistance in using the PAT system. Argonne National Laboratory provided access to the BBN TC2000 computer.

REFERENCES

1. Abraham, S. G., and Hudak, D. E. Compile-time partitioning of iterative parallel loops to reduce cache coherence traffic. *IEEE Trans. on Par. and Dist. Sys.* **2,** 3 (July 1991), 318–328.
2. BBN Advanced Computers, Inc. *Inside the TC2000 Computer.* BBN Advanced Computers, Inc., Cambridge, MA, 1990.
3. BBN Advanced Computers, Inc., *TC2000 Fortran Reference,* BBN Advanced Computers, Inc., Cambridge, MA, 1990.
4. Bromley, M., Heller, S., McNerney, T., and Steele, G., Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation,* 1988, pp. 58–62.
5. Callahan, D., Cocke, J., and Kennedy, K. Estimating interlock and improving balance for pipelined architectures. In *International Conference on Parallel Processing,* 1987, pp. 295–304.
6. Callahan, D., Kennedy, K., and Porterfield, A. Software prefetching. In *Arch. Support for Programming Languages and Operating Systems—IV,* 1991, pp. 40–52.
7. Carr, S., and Kennedy, K. Blocking linear algebra codes for memory hierarchies. In *Proc. SIAM Conference on Parallel Processing for Scientific Computing,* Chicago, IL, December 1989.
8. Carr, S., and Kennedy, K. Compiling scientific code for complex memory hierarchies. In *Proc. Hawaii International Conference on System Sciences,* 1991, pp. 536–544.
9. Cheong, H., and Veidenbaum, A. Compiler-directed cache management in multiprocessors. *IEEE Computer* **23,** 6 (June 1990), 39–47.
10. Cytron, R., Karlovsky, S., and McAuliffe, K. Automatic management of programmable caches. In *International Conference on Parallel Processing,* 1988, pp. 229–238.
11. Farhat, C. A simple and efficient automatic FEM domain decomposer. *Computers and Structures* **28**(5) 579–602, 1988.
12. Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremere, U., Tseng, C., and Wu, M. Fortran D language specification. Tech. Rep. TR90-141, Department of Computer Science, Rice University, Dec. 1990.
13. Gallivan, K., Jalby, W., and Gannon, D. On the problem of optimizing data transfers for complex memory systems. In *ACM International Conference on Supercomputing.* St. Malo, France, 1988, pp. 238–253.
14. Gannon, D., Jalby, W., and Gallivan, K. Strategies for cache and local memory management by global program transformation. *J. Parallel Distrib. Comput.* **5,** 5 (Oct. 1988), 587–616.
15. Gornish, E. H., Granston, E. D., and Veidenbaum, A. V. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ACM International Conference on Supercomputing.* 1990, pp. 354–368.

16. Hudak, D. E., and Abraham, S. G. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *ACM International Conference on Supercomputing*. 1990, pp. 187–200.
17. Hudak, D. E., and Abraham, S. G. Multidimension extensions to adaptive data partitioning. Tech. Rep. CSE-TR-85-91, The University of Michigan, 1991.
18. Knobe, K., Lukas, J., and Steele, G., Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *J. Parallel Distrib. Comput.* **8** (1990), 102–118.
19. Koelbel, C., and Mehrotra, P. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parallel Distrib. Systems*. **2**, 4 (Oct. 1991), 440–451.
20. Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th International Symposium on Computer Architecture*. 1990, 148–159.
21. Mangione-Smith, W., Abraham, S., and Davidson, E. The effects of memory latency and fine-grain parallelism on Astronautics ZS-1 performance. In *Proc. Hawaii International Conference on System Sciences*. 1990, 288–296.
22. Melear, C. The design of the 88000 RISC family. *IEEE Micro* (Apr. 1989), 26–38.
23. Pingali, K., and Rogers, A. Compiling for locality. In *International Conference on Parallel Processing*. 1990, 142–146.
24. Polychronopoulos, C. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems*. Ph.D. thesis, University of Illinois at Urbana–Champaign, Aug. 1986. CSRD Report 595.
25. Ramanujam, J., and Sadayappan, P. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel Distrib. Sys.* **2**, 4 (1991), 472–482.
26. Reed, D. A., Adams, L. M., and Patrick, M. L. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. Comput.* **C36**, 7 (July 1987), 845–858.
27. Smith, K., and Appelbe, W. PAT—An interactive Fortran parallelizing assistant tool. In *International Conference on Parallel Processing*. 1988, 58–62.
28. Tseng, P.-S. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. Ph.D. thesis, Carnegie–Mellon University, Pittsburgh, PA, May 1989.
29. Wolf, M., and Lam, M. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991, pp. 30–44.
30. Zima, H., Bast, H., and Gerndt, M. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Comput.* **6**, (1988) 1–18.

Received September 1, 1991; revised February 28, 1992; accepted April 17, 1992

DAVID E. HUDAK is a Ph.D. student in the Electrical Engineering and Computer Science Department at 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. David Hudak has the B.S. in mathematics from Bowling Green State University, and the M.S. in computer science from the University of Michigan.

SANTOSH G. ABRAHAM is currently an Assistant Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. From 1984 to 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. Santosh Abraham 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.