Compiling for Hierarchical Shared-Memory Multiprocessors

J. D. Martens and D. N. Jayasimha*
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210-1277
(614) 292-1932
martens@cis.ohio-state.edu
jayasim@cis.ohio-state.edu

January 24, 1994

A preliminary version of the first three sections of this paper appeared in the *Proceedings of the 1993 International Conference on Parallel Processing.*

*Both authors were supported in part by the National Science Foundation under grant number NSF CCR-8909189.

Abstract

Latency and synchronization overheads have been identified as two fundamental problems in large scale multiprocessing. Recently, a number of hierarchical shared-memory multiprocessors have been proposed to alleviate these problems. These architectures form a bridge between the shared memory and distributed memory approaches; for example, HSMAs support for consumer-initiated communication is analogous to that of shared-memory multiprocessors because of the presence of a global address space, and HSMA exploitation of data locality is analogous to that of distributed-memory multicomputers because of the presence of an explicitly hierarchical memory. Issues related to compiling for such architectures are investigated in this paper. In particular, three strategies for parallelizing nested iterative loops with subscripts of the form $i \pm k$, where i is a loop index and k is a constant are discussed. The first method uses the abstraction of simple or regular sections to place variables in the hierarchical memory and allows vectorized communication. The second exploits the data dependence graph and in some cases results in less communication than the first. The third is a modification of the second, resulting in a larger number of writes to the hierarchical memory, but allowing lower latency reads.

Key words: parallelizing compilers, iteration space tiling, data space partitioning, hierarchical memory, shared-memory multiprocessing, latency, synchronization.

1 Introduction

In this paper issues related to compiling for shared-memory multiprocessors with multilevel memory hierarchies are investigated. In particular, methods for parallelizing iterative loops (e.g., `for` or `do` loops), and synchronization, communication, and performance issues that arise from parallelization are discussed.

A hierarchical shared-memory multiprocessor architecture (HSMA) is a multiprocessor in which the memories form a *logical* hierarchy, as opposed to the traditional physical hierarchy; in this logical hierarchy some memories are closer to some processors than to others and perhaps some memories are directly accessible by only a subset of the processors. For example, a variable at level i of the memory hierarchy could be shared by a larger number of processors than a variable at level j, where $j < i$. An architecture incorporating a hierarchical memory would have local memory at its lowest level and fully shared global memory at its highest level with a "continuum" in between. Architectures which can be considered HSMAs include explicitly hierarchical machines such as the Tree-structured Hierarchical Memory Multiprocessor (THMM)[15, 23], Cedar[11, 19], Mabbs' Hierarchical Memory Structure[21], Wei and Levy's multilevel hierarchy[33], and the hierarchical multiprocessor of Mahgoub and Elmagarmid[22]. Other architectures which may be considered hierarchical using this definition are non-uniform memory access (NUMA) architectures, such as the BBN Butterfly and shared-memory hypercubes[5, 20]. The use of local and hierarchical memory is not to be confused with the use of caches, since caches form part of the physical hierarchy; in particular, the issue of cache coherence is orthogonal to the issues discussed in this paper.

HSMAs are appealing since they reduce latency and synchronization overheads often associated associated with large-scale multiprocessing[2] (see [15] for a detailed discussion on how different types of communication and synchronization costs are reduced). These architectures form a bridge between the shared-memory and distributed-memory approaches; for example, HSMAs support for consumer-initiated communication is analogous to that of shared-memory multiprocessors because of the presence of a global address space, and HSMA exploitation of data locality is analogous to that of distributed-memory multicomputers because of the presence of an explicitly hierarchical memory. Many parallel algorithms (e.g., matrix multiplication, associative operations, solution of partial differential equations by differencing techniques, prefix computations, barrier synchronization, etc.) have data whose movement behaves in a *structured, hierarchical manner*; such algorithms map well to hierarchical architectures[6, 15, 24, 28, 31, 40]. In many algorithms the various data need not be globally shared, but rather can be *partially shared* among processing elements (PEs)[15]. When a data item is not needed by all PEs, it need not be placed in global memory; rather it is placed at an intermediate level of the memory which is accessible by the subset of PEs requiring the data item. This "intermediate level" may be fewer network hops from the PEs than the global level, thus reducing latency (see [15] for more details).

In the next section a method of parallelizing loops for HSMAs is described; this includes discussion of communication and synchronization primitives, partitioning, and code generation issues. Examples of the application of the method are provided in section 3, along with discussion of related issues. Two alternate parallelization methods are then presented in section 4. This research is compared with related projects in section 5. Finally, the conclusions are presented in section 6.

2 Compiling for HSMAs

It is desirable that compiler support be provided for multiprocessors in general and for HSMAs in particular. If a compiler can automatically restructure a program for a given target machine, then the programmer is free to concentrate more fully on algorithmic issues. Automatic restructuring is helpful in both the initial development and the subsequent maintenance of a program, since the source code is both shorter and at a higher level of abstraction. Compiler support also helps to improve the portability of programs across different multiprocessor architectures. The methods presented concentrate on `for` (or `do`) loops, which are a major source of parallelism and have been the focus of a large body of parallelizing compiler research. These methods may be implemented as part of a compiler, but can also be followed by a programmer as a part of his or her programming methodology.

2.1 Background and Notation

It is assumed that the reader is familiar with standard parallelizing compiler terminology and techniques as discussed in [27, 38]. This paper focuses on nested parallel loops in which subscripts along each dimension of the loop are of the form $i \pm k$, where i is a loop index and k is a constant; it is not necessary that the loops be perfectly nested. The *iteration space* is the set of all values taken on by loop indices. If the loop is nested d levels deep, then the iteration space can be represented by a d-dimensional lattice. This space is related to but distinct from the *data space*, which is the set of array locations written to (or *defined*) or read from (*used*) during the loop's execution. In order to execute a loop in parallel, the iteration space is partitioned into disjoint *tiles* such that the union of all tiles is precisely the iteration space (thus a *tiling* is indeed a partitioning). Each tile is statically mapped to a particular PE. Optimum "shape" and dimensionality of the tiles should be determined experimentally for the target machine and individual program. Tiling has been discussed in depth elsewhere[29, 34, 35, 36, 37]. Associated with each tile is the set of data defined or used within the tile; this set is termed a *footprint*. The footprints are not necessarily disjoint—often footprints of neighboring tiles overlap.

In dataflow analysis, *entry* is used to refer to the definition of data that is used prior to the code fragment under consideration, and *exit* is used to refer to references to data that occur subsequent to the code under consideration.

The following notation is used throughout the rest of this paper:

- The number of PEs is p, numbered $0 \ldots p-1$. The variable m refers to the number of a particular PE, e.g., PE_m.
- The body of a loop consists of a sequence of one or more statements S_i; $use^j_m(L_k)$ is the set of data used by PE_m in the body of loop L_k; j, if present, is an element of the vector of values taken on by the induction variables of the enclosing loops. Alternatively, S_i can be used in place of L_k to indicate data used in S_i. def^j_m is defined similarly.
- lb^d and ub^d are the lower and upper bounds of the d^{th} dimension of a loop nest, respectively.
- lo^d_m and hi^d_m are the lower and upper bounds of the d^{th} dimension of PE_m's tile, respectively.
- Two or more PEs are referred to as *neighbors* if the tiles mapped to the PEs are contiguous. This is not meant to imply that the PEs are nearest neighbors in the underlying interconnection network.

2.1.1 Communication and Synchronization Primitives

In order to write a parallel program or to parallelize a sequential program, support for communication and synchronization among PEs is needed. On an HSMA, a prerequisite for communication is that a variable be placed at a location in the hierarchy accessible by every PE needing the variable, but it is not necessary that the variable be accessible by other PEs. This is one characteristic of a partially-shared variable[15]. Additionally, each partially-shared variable has *state* and *data* fields associated with it. As in [41], the state field is a generalization of the full/empty bit of the Denelcor HEP[17], being an arbitrary integer rather than just a single bit; the state is used for synchronization: producers set the state of a variable to a predetermined value when they write the datum, and consumers wait for the state to reach a particular value before reading the datum. The data field is simply the value of the variable. A function, *commonLevel()*, is provided in order for two or more PEs to agree upon a location at which to communicate. Suppose P is a list of accessing PEs; then the item shared among the PEs in P will be mapped to $commonLevel(P)$. $CommonLevel(P)$ is a machine-dependent function which returns a memory unit efficiently accessible by all PEs in the list P. Note that on some machines a given set of PEs may be able to share data efficiently at more than one memory unit; in these cases it is necessary that $commonLevel()$ provide the same memory unit for each of a set of accessing PEs. Language primitives for communication are also needed; those used here are based upon the atomic instruction proposed by Zhu and Yew[41], with the following form: { address; condition; operation on state; operation on value }. The operational semantics is given in figure 1. Addresses are of the form *variable@memoryUnit*.

```
while not condition do /* busy wait for condition */
    skip
od
operation on address.state
operation on address.value
```

Figure 1: Operational semantics of atomic synchronization instruction.

Note that the semantics of this atomic instruction differs slightly from that proposed by Zhu and Yew, in that the atomic statement always waits for the condition to be satisfied. For convenience, multiple addresses (e.g., a vector) can be specified in one statement if the same operation is to be performed on each address.[1] Initially, all data memory is uninitialized, and all state values are zero.

Finally, for convenience, *nonlocal* is referred to as if it were a specific memory unit in which nonlocal variables can be stored.

2.1.2 Generation of Communication Statements

Producers use atomic statements to set the states of shared variables to loop-specific values (this is similar to what is done in [7] and [16]); the consumer waits for the state to reach the appropriate value. The first parallelization method discussed in this paper, the *basic method*, does not distinguish between array elements shared between just two PEs and elements shared among more than two PEs; as a result, a PE producing a value writes it once for each consumer. This may

[1] In this case, the operation itself is not atomic, but each operand will be read or written in an atomic step.

result in the same value with the same state being written more than once to a particular location. The consumers do not change consumed variables' states, so barriers are needed between loops to ensure that values are consumed before being overwritten. This is discussed in detail later.

One drawback to the basic method is the extra traffic incurred when a producer writes the same value into the same location, with the same key, more than once. Another drawback is that barriers are needed at the end of each parallel loop; this is because the producer has no way of determining whether prior consumers of a value have already obtained the value. Two alternatives are discussed in section 4.

2.2 The Basic Method

The basic method involves the following steps, each of which is later discussed in depth.

1. Parallelizing the loop nest.

2. Performing array dataflow analysis.

3. Partitioning (or tiling) the iteration space.

4. Determining the *use/def* sets (the sets of data read and written, respectively) from within each tile, i.e., determining the footprint of each tile.

5. Determining the intersections of live portions of footprints.

6. Adding communication and synchronization to satisfy dependences due to the intersections of live portions of disparate PEs' *use* and *def* sets.

2.2.1 Parallelizing the Loop Nest

The first step is to parallelize the loop nest. If the loops are perfectly nested, then unimodular transformations[2] may be applied to produce at least one level of parallelism as close as possible to the outermost level of nesting (it is assumed that the nest is more than one level deep, or that it is a non-nested loop in which any dependence cycles can be broken; otherwise the loop must be executed as a doacross, executed sequentially, or rewritten as a recurrence calculation similar to parallel prefix—see [1, 38]). If the loops are not perfectly nested, an attempt is made to obtain outer loop parallelism either directly or via loop interchange. Failing this, an attempt is made to distributed the loop nest and apply unimodular techniques. Outer loop parallelism is preferable since this maximizes the parallel grain size.

If loop distribution was applied to the original loop nest, then to minimize communication it is advantageous to make successive resulting loop nests parallel in the same dimension of the data space.

As an example, consider the program fragment and associated dependence information (produced by the Tiny program restructuring tool[39]) shown in figure 2. Tiny indicates a flow dependence from S_3 to S_4; in parentheses are the direction vector and the distance vector. These vectors each have just one component in this example since the loop nest is just one level deep. In other

[2] A unimodular transformation maps an n-dimensional iteration space into an n-dimensional iteration space, has only integral components, and has determinant ± 1. See [35] for a detailed introduction to the use of unimodular transformations in the context of parallelizing compilers.

```
  2: for i := 1,n do
  3:  a[i] := a[i]*b[i]
  4:  c[i] := a[i-1]*2
  2: endfor

flow dependence 3: --> 4:(<)  (1)
```

Figure 2: Example sequential loop.

examples, an asterisk indicates unavailable (or inapplicable) information. After loop distribution the loop is parallelized as in figure 3. There is still a flow dependence from S_6 to S_9; this indicates

```
L1  5: doall i := 1,n do
    6:  a[i] := a[i]*b[i]
    5: endfor
L2  8: doall i := 1,n do
    9:  c[i] := a[i-1]*2
    8: endfor

flow dependence 6: --> 9
```

Figure 3: Example sequential loop after parallelization.

possible communication between PEs executing these statements. This communication is addressed in subsequent steps of the basic method.

2.2.2 Array Dataflow Analysis

Gross and Steenkiste developed a method for performing interval-based dataflow analysis on arrays[12]. Using their method, it is possible to determine which sections of an array are live at any given point in the program, and the definition which produced that section can be identified. A *section* in this context is a regular section, as defined in section 2.2.4. Conversely, for any definition it is possible to determine at what points later in the execution of the program it will be used.

In the current example, L_2 uses {a[0] from entry, a[1:n-1] from L_1}.

2.2.3 Tiling the Iteration Space, One Tile Per PE

Often it suffices to partition the iterations of the outermost `doall`; however, if the loop can be restructured for multiple levels of parallelism, then better load balancing and lower communication costs can often be obtained by exploiting parallelism in more than just the outermost level. A block partitioning strategy is used throughout this paper, but cyclic and block-cyclic partitions may be investigated as future research (definitions of block, cyclic, and block-cyclic may be in found in [10]). The appeal of a block partition is twofold: (1) if loop alignment is off slightly between successive loops, relatively little additional communication results; (2) the regular and simple section formalisms discussed in sections 2.2.4ff are better suited to block partitions than to cyclic partitions.

In the example of figure 3, the tile will be one-dimensional (a line segment). PE_m executes iterations $lo_m \ldots hi_m, 0 \leq m < p$:

$$\begin{aligned} lo_m &= hi_{m-1} + 1, \text{ where } hi_{-1} = lb - 1 \\ hi_m &= lb - 1 + \left\lceil \frac{(m+1)(ub - lb + 1)}{p} \right\rceil \end{aligned}$$

In this example $lb = 1$ and $ub = n$.

2.2.4 Determining Footprints of Tiles

In order to determine and manipulate the footprints of tiles, it is necessary to represent them in some fashion. The elements of a d-dimensional array can be thought of as being mapped to a d-dimensional cartesian space in which the indices of each array element are used as the coordinates of the corresponding point in d-space. Given such a d-dimensional space, two concise ways of representing subsets of arrays are simple sections[3, 4] and regular sections[13]. A regular section is a rectangular portion of an array, with each regular section boundary parallel to a coordinate axis. Simple sections are a bit more flexible in that they also allow diagonal boundaries at 45 degree angles to the coordinate axes. Since data represented by regular sections are easier to manipulate from both expositional and computational points of view, the regular section abstraction will be assumed below. Note that sections are not used in the dependence analysis used for parallelization—sections are only used for determining required communication in an *already parallelized* program. Thus, at worst, the use of sections to conservatively approximate required communication incurs extra communication.

In the example of figure 3, data used and defined by PE_m, i.e., PE_m's footprint, is represented by the following regular sections:

$$\begin{aligned} def_m(L_1) &= \{a[lo_m : hi_m]\} \\ use_m(L_1) &= \{a[lo_m : hi_m], b[lo_m : hi_m]\} \\ def_m(L_2) &= \{c[lo_m : hi_m]\} \\ use_m(L_2) &= \{a[lo_m - 1 : hi_m - 1]\} \end{aligned}$$

2.2.5 Intersections of Live Portions of Footprints

Once the footprints are known and the dataflow analysis has been performed, it is possible to determine where partially-shared data must be sent and where partially-shared data can be obtained from. This is determined by intersecting the live sets with the footprints. However, determining which footprints must be intersected may be nontrivial.

As explained above, each tile is statically mapped to a PE. Although the tiles are disjoint, the corresponding footprints may not be. The intersections of disparate footprints must be found. Simple sections are closed under intersection, i.e., the intersection of two simple sections can be precisely represented as a simple section. Likewise, regular sections are closed under intersection. With p PEs, each PE determines the relevant intersections between its footprint and those of the other $p-1$ PEs. Any array element that is within a nonempty intersection of sections must be mapped to the hierarchy.

Without static knowledge of p and the loop bounds, a considerable portion of this calculation may have to occur at runtime. With a large number of PEs this is not desirable, so a less expensive technique is needed.

If dependence distances (in the original loop prior to restructuring) are available and these distances are small relative to the footprint sizes, then all communication required for correct execution of a tile is between PEs with neighboring tiles. This condition can be formalized as:

$$\bigwedge_{i=1}^{d}(dd_i \leq \tfrac{n_i}{\sqrt[d]{p}}) \tag{1}$$

where dd_i is the displacement of the dependence vector in the i^{th} dimension (i.e., the dependence distance in the i^{th} dimension), d is the dimensionality of the tile, n_i is the total number of iterations in the i^{th} dimension, and p is the number of PEs.

Theorem 2.1 *Condition (1) holds iff all communication due to a particular dependence is between contiguous (at edges or corners) pairs of tiles.*

Proof: If the dependence distance in the i^{th} dimension, dd_i, is greater than the width of the tile in that dimension, then its source and sink need not be in adjacent tiles, and for iterations at the tile boundary of the i^{th} dimension the source and sink will not be in adjacent tiles. If, on the other hand, the dependence distance in the i^{th} dimension is no more than the width of the tile in that dimension, then the source and sink are no farther apart than a tile width, and thus must reside in adjacent (or the same) tiles.

There are p tiles, and a block partition is assumed. Thus there are $\sqrt[d]{p}$ tiles per dimension of the iteration space, and the width of a tile in the i^{th} dimension is $\frac{n_i}{\sqrt[d]{p}}$ iterations. So, any communication in the i^{th} dimension will be among adjacent tiles iff $dd_i \leq \frac{n_i}{\sqrt[d]{p}}$, and all communication will be among adjacent tiles iff this is true for all dimensions. □

Often not all of dd, n_i, and p will be known at compile time (d is always known statically since the number of dimensions of the partition corresponds to the number of levels of loop parallelism), in which case the compiler will have to generate multi-version code: one version for "short" communication satisfying condition (1), and another version for cases in which condition (1) is not satisfied.

In figure 3, if $n \geq p$, then condition (1) is satisfied, so communication will be with PEs executing neighboring tiles. For $n < p$, high-numbered PEs will have no work, so communication among *active* PEs is still between neighboring tiles. Since this is a one-dimensional partition, the PEs with tiles neighboring those of PE_m will be $\text{PE}_{m\pm1}$. Considering only legal dependences, the data communicated will thus be a subset of:

$$def_m(L_1) \cap use_{m+1}(L_2),$$

which is just the element $a[hi_m]$. So, after loop L_1, PE_m must write $a[hi]$ to $commonLevel(m, m+1)$. Performing a complementary analysis for PE_m on L_2, we see that PE_m must read $a[lo_m - 1]$ from $commonLevel(m, m-1)$ prior to executing loop L_2.

The example of figure 3 when tiled as in section 2.2.3 satisfies condition (1); thus the values used by processor m within L_2 that are both generated within L_1 and still live are determined by calculating $def_{m-1}(L_1) \cap use_m(L_2) \cap \{a[0] \text{ from entry}, a[1 : n-1] \text{ from } L_1\}$, which simplifies to $\{a[lo_m]\}$.

2.2.6 Communication and Synchronization

Since the parallel loops under consideration—after restructuring—are doall loops, there will be no loop-carried dependences. This simplifies matters considerably. An initialization phase is placed at the beginning of each doall. In this initialization phase, all data needed by the PE for execution of its entire tile (i.e., all live data used within the tile that does not already reside within the PE) is obtained; data already present at the PE need not be fetched. Then the PE executes its tile. After the tile has been executed, all modified data that will be subsequently needed by other PEs is written to appropriate levels of the hierarchy; this is termed loop finalization. This scheme of fetching needed data, executing, and then sending live data elsewhere is also used by Ramanujam and Sadayappan[29]. In Cedar terminology the initialization and the finalization are referred to as the *preamble* and the *postamble*, respectively[9].

The approximate code generated is given in figure 4 (for simplicity, boundary conditions and the ceiling function of the lo_m, hi_m calculations have been omitted). Except within atomic communication statements, all variable names refer to local addresses.

```
lo := 1 + m * n div p
hi := (m + 1) * n div p
for i := lo,hi do
  a[i] := a[i]*b[i]
endfor
Barrier
{a[hi]@commonLevel(m, m+1); true; state := 1; value := a[hi]}  /* send a[hi] to PE m+1 */
{a[lo-1]@commonLevel(m, m-1); state=1; nop; a[lo-1] := value}  /* get a[lo-1] from PE m-1 */
for i := lo,hi do
  c[i] := a[i-1]*2
endfor
```

Figure 4: Example of figure 3 with communication added. Every PE executes the code in this figure; m is the processor number.

Note that a section can be viewed as a vector of vectors, and thus communication of intersections of sections is easily vectorized. Therefore had the example required communication of more than a single array element per PE pair, that communication would have been expressed as a vector.

3 Examples

The method is best illustrated through the presentation of examples. The examples chosen reflect commonly occurring forms of parallelism: inner and outer loop parallelism, wavefront parallelism, and orthogonal parallelism.

3.1 A Doubly-Nested Loop

3.1.1 The Loop

Consider the loop in figure 5.

```
for i := 1 to n do
      for j := 1 to n do
            a[i+1, j] := f(a[i, j])
            a[i, j+1] := g(a[i, j+1])
            a[i+1, j-1] := h(a[i, j])
      endfor
endfor
```

Figure 5: Doubly-nested loop with dependences carried by outer loop.

3.1.2 Step One

After statement reordering and loop distribution, the loop can be parallelized as in figure 6.

```
   for i := 1 to n do
L1   doall j := 1 to n do
S1         a[i, j+1] := g(a[i, j+1])
     endfor
L2   doall j := 1 to n do
S2         a[i+1, j] := f(a[i, j])
     endfor
L3   doall j := 1 to n do
S3         a[i+1, j-1] := h(a[i, j])
     endfor
   endfor
```

Figure 6: Doubly-nested loop after parallelization.

3.1.3 Step Two

With live variable analysis it is possible to determine which definition of an array element is used at any given point in the program. For example, in the loop currently under consideration, the data used in L_1 is shown in figure 7 as a subset of 2-space. Each rectangular region corresponds to a regular section, and each is labeled with the source of the data needed in S_1. This can be expressed as in figure 8.

3.1.4 Step Three

The next step is to tile the iteration space using block partitions. PE_m, $0 \leq m < p$, executes the code shown in figure 9.

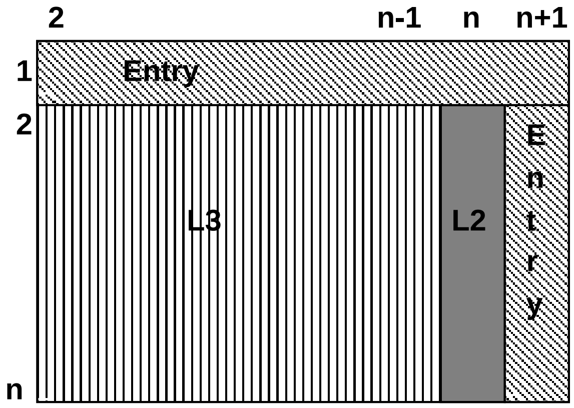

Figure 7: Sources of live data.

L_1 uses { a[1, 2:n+1] from Entry,
a[2:n, 2:n-1] from L_3,
a[2:n, n] from L_2,
a[2:n, n+1] from Entry }

Figure 8: Regular sections used in L_1.

```
     for i := 1 to n do
L1     for j := lo_m to hi_m do
S1          a[i, j+1] := g(a[i, j+1])
       endfor
       Barrier
L2     for j := lo_m to hi_m do
S2          a[i+1, j] := f(a[i, j])
       endfor
       Barrier
L3     for j := lo_m to hi_m do
S3          a[i+1, j-1] := h(a[i, j])
       endfor
       Barrier
     endfor
```

Figure 9: Code fragment of figure 6 after tiling.

3.1.5 Step Four

In the fourth step, it is necessary to determine the footprints defined and referenced within each tile. The footprints relevant to the data references in L_1 are represented as regular sections in figure 10.

1. $use^i_m(L_1) = \{a[i, lo_m + 1 : hi_m + 1]\}$
2. $def^{1:i-1}_m(L_2) = \{a[2 : i, lo_m : hi_m]\}$
3. $def^{1:i-1}_m(L_3) = \{a[2 : i, lo_m - 1 : hi_m - 1]\}$

Figure 10: Footprints of tiles referenced by PE_m in L_1.

3.1.6 Step Five

Now the live portions of intersections of relevant footprints are determined. Prior to L_1, PE_m must fetch the data as shown in figure 11. The first two regular sections in figure 11 are the data

1. $use^i_m(L_1) \cap \{a[1, 2 : n + 1]\}$ from Entry;
2. $use^i_m(L_1) \cap \{a[2 : n, n + 1]\}$ from Entry;
3. $\forall m' : 0 \leq m' < p \wedge m' \neq m :$
   $use^i_m(L_1) \cap def^{1:i-1}_{m'}(L_2) \cap \{a[2 : n, n]\}$ from commonLevel(m, m');
4. $\forall m' : 0 \leq m' < p \wedge m' \neq m :$
   $use^i_m(L_1) \cap def^{1:i-1}_{m'}(L_3) \cap \{a[2 : n, 2 : n - 1]\}$
   from commonLevel(m, m');

Figure 11: Live portions of sections used in L_1 of the doubly-nested loop.

needed in L_1 that are not written elsewhere in the loop nest. This is simply each section of live data from the entry node intersected with the data used over the course of L_1 by PE_m in iteration i of the outer, sequential loop.

The third section is data used by PE_m in L_1 intersected with data generated by other PEs in earlier iterations of the outer loop intersected with data live from L_2 at that point.

The fourth section is similar to the third, except that the fourth section is live data used in L_1 that is generated within L_3.

If the loop bounds and number of processors are known statically, the compiler can simplify the preceding as in figure 12. The first three sections in figure 12 correspond directly to the first three sections in figure 11. Regular sections four through six in figure 12 are derived from the fourth section in the preceding step. The reason this regular section is now represented as three is due to the fact that multiple cases must be considered: one case for when condition one is satisfied, and one for each possible communication when condition one is not satisfied. When condition one is not satisfied and the dependence distance is d, then there will be potential communication between PE_m and d of its neighbors, so there will be d sections; in this case, $d = 2$.

1. { a[1, $lo_m : hi_m$] } from Entry if $i = 1$
2. { a[$i, n+1$] } from Entry if $i \neq 1 \wedge n = hi_m$
3. { a[i, n] } from commonLevel(m, m+1) if $hi_m = n$
4. { a[i, lo_{m+1} :min($hi_m + 1, hi_{m+1} - 1$] } from commonLevel(m, m+1) if (condition one is satisfied) $\wedge\ j < n\ \wedge\ i > 1$
5. { a[$i, lo_{m+1} - 1 : hi_m + 1$]} from commonLevel(m, m+1) if (condition one is not satisfied) $\wedge\ j < n\ \wedge\ i > 1$
6. { a[$i, lo_{m+2} - 1 : hi_m + 1$]} from commonLevel(m, m+2) if (condition one is not satisfied) $\wedge\ j < n\ \wedge\ i > 1$

Figure 12: The sections of figure 11 after simplification.

3.2 Matrix Multiplication

Consider the matrix multiplication program in figure 13. The i and j loops can execute in

```
2: for i := 1,n do
3:  for j := 1,n do
4:   c[i, j] := 0
5:   for k := 1,n do
6:    c[i, j] := c[i, j]+a[i, k]*b[k, j]
5:   endfor
3:  endfor
2: endfor

flow dependence 4: --> 6:(=,=)  (0,0)
output dependence 4: --> 6:(=,=)  (0,0)
anti dependence 6: --> 6:(=,=,<)  (0,0,*)
flow dependence 6: --> 6:(=,=,<)  (0,0,*)
output dependence 6: --> 6:(=,=,<)  (0,0,*)
```

Figure 13: Matrix multiplication—sequential.

parallel with no other changes, since there are no dependences carried by these two loops, as shown in figure 14.

Array dataflow analysis shows that at S_{11} in figure 14 (line eleven in the figure), a[1:n, 1:n] and b[1:n, 1:n] are needed from entry, and when $k = 1$, c[i, j] is obtained from S_{11}.

If the outer two loops execute in parallel and the inner loop executes sequentially, then the iteration space can be partitioned in rectangular blocks (a block partition in each dimension). Let $row(m) = m$ div $\sqrt{p}$ and $col(m) = m \bmod \sqrt{p}$. Then the bounds of PE$_m$'s partition can be

```
  7: doall i := 1,n do
  8:  doall j := 1,n do
  9:   c[i, j] := 0
 10:   for k := 1,n do
 11:    c[i, j] := c[i, j]+a[i, k]*b[k, j]
 10:   endfor
  8:  endfor
  7: endfor

flow dependence 9: --> 11:(=,=)  (0,0)
output dependence 9: --> 11:(=,=)  (0,0)
anti dependence 11: --> 11:(=,=,<)  (0,0,*)
flow dependence 11: --> 11:(=,=,<)  (0,0,*)
output dependence 11: --> 11:(=,=,<)  (0,0,*)
```

Figure 14: Matrix multiplication—parallel.

expressed for each dimension as:

$$\begin{aligned}
lo_m^1 &= lb^1 + row(m) \times \frac{ub^1 - lb^1 + 1}{\sqrt{p}} \\
hi_m^1 &= lo_m^1 + \frac{ub^1 - lb^1 + 1}{\sqrt{p}} - 1 \\
lo_m^2 &= lb^2 + col(m) \times \frac{ub^2 - lb^2 + 1}{\sqrt{p}} \\
hi_m^2 &= lo_m^2 + \frac{ub^1 - lb^1 + 1}{\sqrt{p}} - 1
\end{aligned}$$

The the use/def sets for statements nine and eleven can be expressed as:

$$\begin{aligned}
def(S_9) &= \{c[lo_m^1 : hi_m^1, lo_m^2 : hi_m^2)\} \\
def(S_{11}) &= \{c[lo_m^1 : hi_m^1, lo_m^2 : hi_m^2)\} \\
use(S_{11}) &= \{a[lo_m^1 : hi_m^1, 1 : n), b[1 : n, lo_m^2 : hi_m^2)\}
\end{aligned}$$

Since the i and j loops carry no dependences, no communication is needed during execution of the loop. Beforehand, however, the initial partition must place elements of a and b where needed, implying that each element of a or b will be replicated $\sqrt{p}$ times.

3.3 Inner Loop Parallelism

Consider the loop nest in figure 15. Once the $i2$ loop is distributed, the $i2$, j, and k loops can execute in parallel, as in figure 16.

```
  5: for i1 := 1,n1 do
  6:  for i2 := 1,n2 do
  7:   for j := 1,n3 do
  8:    a[i1, i2+1, j] := b[i1, i2, j]*d[i1, i2, j]
  7:   endfor
 10:   for k := 1,n3 do
 11:    b[i1+1, i2, k] := a[i1, i2, k+1]+c[i1, i2, k]
 10:   endfor
  6:  endfor
  5: endfor

flow dependence 8: --> 11:(=,<)  (0,1)
flow dependence 11: --> 8:(<,=)  (1,0)
```

Figure 15: Inner loop example.

```
  5: for i1 := 1,n1 do
  6:  doall i2 := 1,n2 do
  7:   doall j := 1,n3 do
  8:    a[i1, i2+1, j] := b[i1, i2, j]*d[i1, i2, j]
  7:   endfor
  6:  endfor
 11:  doall i2 := 1,n2 do
 12:   doall k := 1,n3 do
 13:    b[i1+1, i2, k] := a[i1, i2, k+1]+c[i1, i2, k]
 12:   endfor
 11:  endfor
  5: endfor

flow dependence 8: --> 13:(=)  (0)
flow dependence 13: --> 8:(<)  (1)
```

Figure 16: Inner loop parallelized.

Array dataflow analysis shows that at S_8, b[1, 1:n2, 1:n3] and d[1:n1, 1:n2, 1:n3] are needed from Entry, b[2:n1, 2:n2, 2:n3] are needed from S_{13}, and a[1:n1, 2:n2, 2:n3] are defined and subsequently used at S_{13}.

At S_{13}, the following sections are used: {a[1:n1, 2:n2, 2:n3] from S_8, a[1:n1, 1, 2:n3+1] from Entry, a[1:n1, 2:n2, n3+1] from Entry, c[1:n1, 1:n2, 1:n3] from Entry}. b[2:n1, 1:n2, 1:n3] is defined and subsequently used in S_8.

The dependences are between S_8 and S_{13}, so the data used and defined in those statements must be determined:

$$\begin{aligned}
use_m(S_8) &= \{b[i1, lo_m^2 : hi_m^2, lo_m^3 : hi_m^3), d[i1, lo_m^2 : hi_m^2, lo_m^3 : hi_m^3]\} \\
def_m(S_8) &= \{a[i1, lo_m^2 + 1 : hi_m^2 + 1, lo_m^3 : hi_m^3)\} \\
use_m(S_{13}) &= \{a[i1, lo_m^2 : hi_m^2, lo_m^3 + 1 : hi_m^3 + 1], c[i1, lo_m^2 : hi_m^2, lo_m^3 : hi_m^3]\} \\
def_m(S_{13}) &= \{b[i1 + 1, lo_m^2 : hi_m^2, lo_m^3 : hi_m^3)\}
\end{aligned}$$

Since the dependence distances are zero or one, condition one is satisfied. Thus communication will be with neighbors in the partition. In two dimensions, a tiling may have up to eight neighbors—one along each boundary and one at each vertex, as in the sixteen PE tiling shown in figure 17a. A PE

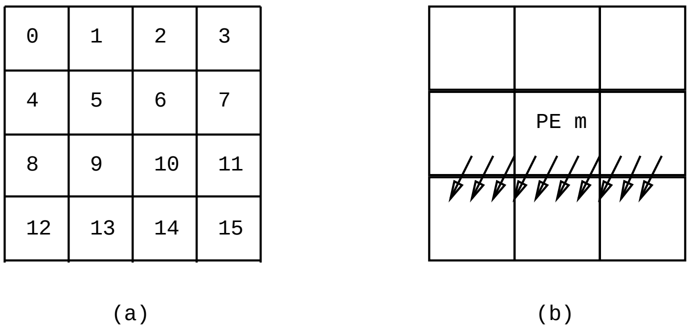

Figure 17: Two dimension block partition.

can determine whether it has an east neighbor by evaluating $(m + 1) \bmod \sqrt{p}$; if this is nonzero, then the east neighbor exists and is PE_{m+1}. Likewise, PE_m has a west neighbor iff $m \bmod \sqrt{p}$ is nonzero, and the west neighbor is PE_{m-1}. If $m \geq \sqrt{p}$, then PE_m has a north neighbor, $\mathrm{PE}_{m-\sqrt{p}}$. Finally, iff $m < p - \sqrt{p}$, PE_m has a south neighbor, $\mathrm{PE}_{m+\sqrt{p}}$.

Diagonal neighbors are determined by conjoining the above predicates, and composing their respective arithmetic expressions. For example, PE_m has a southeast neighbor iff it has both a south and an east neighbor, and if this neighbor exists, it will be $\mathrm{PE}_{m+\sqrt{p}+1}$. The reader can verify that similar reasoning shows that tile five in a two-dimensional tiling has neighbors zero, one, two, four, six, eight, nine, and ten, as in figure 17a.

Now the data that each PE must read and write into the hierarchy must be determined by taking the intersections of $def_m(S_8)$ and $use_k(S_{13})$ for each k such that k is a neighbor of m in the partition.

In order to determine these intersections, values for lo_m and hi_m in each dimension must be found for each PE. These will be identical to those obtained in the matrix multiplication example, except that the first and second dimensions in that example are called the second and third in this one, respectively. Note that except for obtaining data from Entry, the array dataflow analysis is not needed in this example. This is because none of the data defined in the fragment is killed within the fragment.

Looking at the dependences, it can be seen that within an iteration of the $i1$ loop, each element of a defined in S_8 will be used in S_{13}, with the partition translated down (in the second dimension) and to the left (in the third dimension), as in figure 17b. This can be derived by intersecting $def_m(S_8)$ with each of $use_{m-1}(S_{13})$, $use_{m+\sqrt{p}}$, and $use_{m+\sqrt{p}-1}(S_{13})$, which evaluate to $a[i1, hi^2_m + 1, lo^3_m : hi^3_m)$, $a[i1, lo^2_m + 1 : hi^2_m + 1, lo^3_m)$, and $a[i1, hi^2_m + 1, lo^3_m)$, respectively.

Each of these sections must be written to a level of the hierarchy and corresponding reads must be generated. The level is obtained via a call to commonLevel(). The dependence from S_{13} to S_8 is carried by the sequential loop, and thus generates no communication.

3.4 Five Point Stencil

Consider the five point stencil code in figure 18. After skewing the inner loop by a factor of one

```
  2: for i := 1,n1-2 do
  3:  for j := 1,n2-2 do
  4:   a[i, j] := (a[i, j]+a[i-1, j]+a[i+1, j]+a[i, j-1]+a[i, j+1])/5
  3:  endfor
  2: endfor

anti dependence 4: --> 4:(<,=)  (1,0)
anti dependence 4: --> 4:(=,<)  (0,1)
flow dependence 4: --> 4:(<,=)  (1,0)
flow dependence 4: --> 4:(=,<)  (0,1)
```

Figure 18: Five point stencil.

with respect to the outer loop and performing loop interchange, we obtain inner loop parallelism as shown in figure 19.

Since this is inner loop parallelism, the code generated within the body of the j loop treats j almost as a constant. The iteration space partition is single dimensional since there is only one level of parallelism. The endpoints of PE$_m$'s tile in the j^{th} iteration, lo_m and hi_m, are defined as:

$$\begin{aligned} lo_m &= hi_{m-1} + 1, \text{ where } hi_{-1} = lb - 1. \\ hi_m &= lb - 1 + \left\lceil \frac{(m+1)(ub - lb + 1)}{p} \right\rceil, \end{aligned}$$

where $lb = max(i, j - n2 + 2)$ and $ub = min(n1 - 2, j - 1)$.

In iteration j of the outer loop, PE$_m$ defines $def^j_m = a[lo_m : hi_m, j - lo_m : j - hi_m]$, and uses $use^j_m =$

$$\begin{aligned} a[lo_m : hi_m, j - lo_m : j - hi_m] \quad &\cup \\ a[lo_m - 1 : hi_m - 1, j - lo_m : j - hi_m] \quad &\cup \end{aligned}$$

```
  3: for j := 1+1,n2-2+(n1-2) do
  2:  doall i := max(1, j-(n2-2)),min(n1-2, j-1) do
  4:   a[i, j-i] := (a[i, j-i]+a[i-1, j-i]+a[i+1, j-i]+a[i, j-i-1]+a[i, j-i+1])/5
  2:  endfor
  3: endfor

anti dependence 4: --> 4:(<,<)  (1,1)
anti dependence 4: --> 4:(<,=)  (1,0)
flow dependence 4: --> 4:(<,<)  (1,1)
flow dependence 4: --> 4:(<,=)  (1,0)
```

Figure 19: Five point stencil after skewing and interchange.

$$
\begin{array}{ll}
a[lo_m + 1 : hi_m + 1, j - lo_m : j - hi_m] & \cup \\
a[lo_m : hi_m, j - lo_m - 1 : j - hi_m - 1] & \cup \\
a[lo_m : hi_m, j - lo_m + 1 : j - hi_m + 1] & \cup \\
 & = \quad a[lo_m - 1 : hi_m + 1, j - hi_m + 1 : j - lo_m - 1]
\end{array}
$$

Thus, the initial partition should place use_m in each PE $(0 \leq m < p)$ for $j = 2$. Before beginning its tile of iteration j of the outer loop (for $j > 2$), PE$_m$ reads $def^{j-1}_{m-1} \cap use^j_m$ from commonLevel$(m, m-1)$ and $def^{j-1}_{m+1} \cap use^j_m$ from commonLevel$(m, m+1)$. After completing its tile, PE$_m$ writes $def^j_m \cap use^{j+1}_{m-1}$ to commonLevel$(m, m-1)$ and $def^j_m \cap use^{j+1}_{m+1}$ to commonLevel$(m, m+1)$.

Note that the union of the sections arising from the array references on the right hand side is inexact since array elements not actually needed by the referencing PE are included in the intersection. This implies that the referencing PE will obtain extra data, causing extra communication. Furthermore, more than one PE may need a particular array element; when this occurs, it is quite possible that the PEs will share the array element at the same memory unit, implying that the writing PE will place the shared array element in the memory unit more than once. This is discussed further below.

3.5 Orthogonal Parallelism

It is sometimes convenient or even necessary to parallelize successive loop nests such that they execute in parallel along different dimensions of one or more arrays. Consider the loop nests of figure 20.

One possible parallelization is shown in figure 21. This parallelization has the advantage that both loop nests are parallel in the same dimension of the array a, but—as in any wavefront parallelization—the number of iterations of the inner, parallel loop varies between different iterations of the outer, sequential loop and every iteration of the outer loop entails communication among the PEs of values calculated during the prior iteration of the inner loop.

Thus, consider instead an implementation such as that in figure 22. The average amount of parallelism (essentially the inner loop's mean trip count) is higher here than in figure 21, and load balancing is likely to be better in the version of figure 22 than in that of figure 21. On the other hand, the total communication required in the program fragment in figure 22 is higher.

```
2: for i := 1,n do
3:  for j := 1,n do
4:   a[i, j] := a[i, j-1]
3:  endfor
2: endfor
7: for i := 1,n do
8:  for j := 1,n do
9:   a[i, j] := a[i+1, j]
8:  endfor
7: endfor
```

Figure 20: A pair of loop nests.

```
2: doall i := 1,n do
3:  for j := 1,n do
4:   a[i, j] := a[i, j-1]
3:  endfor
2: endfor
7: for j := 1+1,n+n do
8:  doall i := max(1, j-n),min(n, j-1) do
9:   a[i, j-i] := a[i+1, j-i]
8:  endfor
7: endfor
```

Figure 21: A possible parallelization.

```
2: doall i := 1,n do
3:  for j := 1,n do
4:   a[i, j] := a[i, j-1]
3:  endfor
2: endfor
7: for i := 1,n do
8:  doall j := 1,n do
9:   a[i, j] := a[i+1, j]
8:  endfor
7: endfor
```

Figure 22: Another parallelization.

Assume that the second parallelization is chosen. From array dataflow analysis, it can be determined that S_4 uses a[1:n, 0:n-1] from Entry and that S_4 defines a[1:n, 1:n]. S_9 uses {a[2:n, 1:n] from S_4, a[n+1, 1:n] from Entry} and defines a[1:n, 1:n].

Once the loops are tiled, in S_4, PE$_m$ defines $a[lo^1_m : hi^1_m, 1 : n]$ and uses $a[lo^1_m : hi^1_m, 0 : n-1]$. In S_9, PE$_m$ uses $a[2 : n+1, lo^2_m : hi^2_m]$ and defines $a[1 : n, lo^2_m : hi^2_m]$. Note that the parallelism is not in the same dimension between the two loop nests, and thus the communication can be expected to be significant. Furthermore, without dependence distances from S_4 to S_9, condition (1) cannot be satisfied, so it must be assumed that every PE will communicate with every other PE, and that the precise elements communicated cannot be determined at compile time without precise knowledge of p and the loop bounds. Thus, each PE will have to derive communication at runtime.

First code is generated that determines what array elements each PE must write into the memory hierarchy. This is inserted between the two loop nests, and is shown in figure 23.

```
for i:= 0 to p-1 do
      if i ≠ m then
            store a[lo^1_m : hi^1_m, 1 : n] ∩ a[2 : n + 1, lo^2_i : hi^2_i] at commonLevel(i, m)
      fi
od
```

Figure 23: Searching for destinations of writes.

Then we generate code to read needed elements from the hierarchy. This code, shown in figure 24, will immediately follow the code in figure 23.

```
for i:= 0 to p-1 do
      if i ≠ m then
            fetch a[2 : n + 1, lo^2_m : hi^2_m] ∩ a[lo^1_i : hi^1_i, 0 : n − 1] at commonLevel(i, m)
      fi
od
```

Figure 24: Searching for sources of reads.

The cost of performing the intersections in each of these loops is $2pd$ comparisons, where d is the number of dimensions in the partition. Since d is bounded from above by the level of loop nesting, it is effectively a small constant. In practice, $p \leq n$, so if the cost of each of the loops in figures 23 and 24 is amortized over the execution time of the loops in figure 22, they add O(1) overhead per iteration.

3.6 Discussion

Various issues that arise with the method just presented are discussed in this subsection. These include issues of correctness, applicability to other architectures, and how communication between PEs and the hierarchical memory can be achieved in practice.

3.6.1 On the Correctness of Reading and Writing Sections

Lemma 3.1 *Within a* `doall` *loop, there is only one write to any particular array element, but there may be several reads. Thus, the set of elements written within a* `doall` *(arising from the union of multiple sections) need not be approximated by an inexact section, though the set of elements read may be approximate. This does not cause correctness problems.*

Proof: The use set for a particular array from within a particular tile may be an approximation since it is determined by finding the union of (possibly) several simple or regular sections. Sections are not closed under union, so a somewhat larger section is used to approximate the union. This implies that PEs will occasionally attempt to read data that they will not actually use. This "extra" data will be available, since each producer intersects its *def* set with all em use sets of potential consumers of its data; since the same analysis is used to generate code for both the consumers and the producers, they will agree on reading and writing PEs, and the producers will make the data available to consumers, even though some of the data may not actually be used.

The def sets calculated by writing PEs are exact since each def set represents the writes performed by a single PE within a `doall` loop. Our tiles are chosen to be sections, so each def set must be a section. □

3.6.2 Intersections

If there are multiple reads of an array element, the basic method will result in multiple writes of the value into the hierarchy. If two of these writes happen to be to the same level of the hierarchy, then the second will overwrite the first. It is possible to consider the second write to be extraneous.

Lemma 3.2 *If only pairwise intersections of sections are performed, extraneous writes to the hierarchy can occur. This does not cause correctness problems.*

Proof: Suppose w is an extraneous write. Then there are two possible cases to consider: w overwrites data needed by a consumer, and w does not overwrite data needed by a consumer.

It is important to note that all writes are performed from within `doall` statements, so that only one PE will be writing to any given location over the course of a given loop.

If w does not overwrite anything needed by a current consumer, then no harm can be done. There will be an unused datum in the hierarchy with state set to the ID of the current loop. If the location was used by a consumer in a prior loop, that consumer has already accessed the location (guaranteed by the use of barriers) and what is written now is immaterial. If the location is used by a consumer in a subsequent loop, then there is no danger of confusing the current value with the future (correct) value, since the states will differ between different loops.

If w does overwrite a value needed by a current consumer, then it writes the same value with the same state as any other write that can possibly occur within this loop, so again no harm can be done. □

3.6.3 Applicability to NUMA Architectures

Suppose that the shared memory is distributed among the PEs, as in the BBN Butterfly, Dash, the IBM RP3, or PASM. Thus, every shared location is local to some PE.

The method maps directly to such a NUMA architecture, since it identifies information which is needed locally by each processor along with necessary communication. The primary difference between NUMA and HSMA implementations of this method would be in the implementation of *commonLevel*(). In an HSMA, *commonLevel*(L) returns a memory unit which is efficiently accessible by each PE in the list L. The same is true in a NUMA architecture, though in the NUMA "efficiently" may be more difficult to define. In a typical NUMA machine *commonLevel*(L) would place the item in one of the processors in the list L. Ideally, to reduce the cost of busy waiting, the item would be placed in a consumer rather than the producer[25, 26]; this may require an additional parameter to *commonLevel*().

Some coherence problems may be caused by a naive implementation of PEs writing into other PEs' memories; this writing will occur due to the fact that memory local to one PE will be "in the hierarchy" with respect to another PE. This problem can be overcome by either partitioning local memory into shared and private portions[3] or by associating state information with local variables in addition to nonlocal variables.

3.6.4 On Communication and Synchronization

There are a number of different options in providing hardware support for communication and synchronization. Among the options include multithreaded (or "latency-tolerant") PEs, support for vectorized reads and writes, and minimal support, e.g., forcing all reads and writes to be blocking.

Assume that the network is pipelined, so that each PE can have more than one outstanding request (or packet) in the network at a time. Also assume that the time a packet takes to travel from a PE to a memory unit (or vice versa) is at least proportional to the number of intervening switch stages; on a conventional multistage interconnection network, this is $\Omega(\log p)$.

If, on the other hand, the PE supports vector reads and writes, reading a vector from within a single thread will take time $\Omega(v + \log p)$. With proper hardware support, useful computation can be overlapped with this communication. A major difference, then, between the vector and the multithreaded approaches is how the PE is kept busy. A multithreaded PE needs v cycles of CPU time to dispatch v read or write requests, whereas a vector computer can issue the same v requests in $O(1)$ CPU cycles. Thus the vector CPU should be able to perform more useful work in the presence of communication. On the other hand, communication is likely to be somewhat burstier in the vector case, suggesting the contention may be more of a problem; extra hardware or software support is needed to resolve references to fetched data that may not have arrived yet; and some communication patterns are not amenable to vectorization. Finally, a multithreaded PE could also support vector communication.

Since communication in this method is based on sections, and these sections can be thought of as being made up of rows or columns of vectors, it makes sense to prefer vectorized communication in this context. Many multiprocessors, however, do not provide hardware support for vectorized communication. Therefore methods that are more appropriate for finer-grain communication are presented in the next section.

4 The Alternate Method

A method for parallelizing loop nests was presented above; this is referred to as the *basic method.* Two shortcomings of the basic method are that it is possible that multiple writes of the same item to the same location can occasionally occur (as in the five point stencil example of section 3.4), and that barriers are needed between successive `doall` loops. In this section a method is presented

[3] This need not be a physical, hardware-imposed partition. Such a partition is easily achieved in software.

which reduces both the amount of data written and the number of barriers required; this is referred to as the *alternate method.*

Assume a program in which all subscripts are of the form $i \pm k$, where i is a loop index and k is an integer constant; also suppose there are no conditionals. Then it is straightforward to determine from the dependence graph exactly which statements use the data written by any particular assignment statement. Each producer (the LHS of an assignment statement) is a source in the dependence graph, and each consumer (a reference in the RHS of an assignment statement) is a sink in the dependence graph. For each source, all outgoing edges must be traversed (at compile-time) to find the set of consumers; from this is determined what sharing will take place. For each consumer, it is necessary to traverse the incoming edge back to the producer, and then traverse the outgoing edges to find the set of consumers sharing that value—from this set and the PE of the producer the level of the hierarchy at which sharing is to occur can be determined (see figure 25). These sets are then examined at run-time[4] to determine at what level of the hierarchy sharing can occur; note that there is no sharing unless there is a loop-carried dependence, so we restrict our graph for these purposes to vertices and edges involved in loop-carried flow dependences. This is similar to Das et al.'s bipartite runtime dependence graph (BRDG)[8], except that—due to the simpler class of loops considered here—our dependence graph is built and (mostly) analyzed at compile-time.

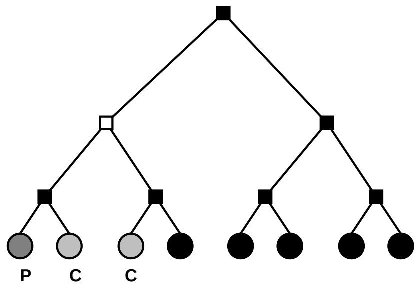

Figure 25: Producer-consumer sharing. Leaves represent PEs and interior nodes hierarchical memory units. A producer (P) and two consumers (C) share data at a node in the second level of the hierarchy (white box). Note that each consumer must know where the producer and the other consumers reside. For example, if a PE in the right subtree were to need the value calculated by the producer, then the data sharing would necessarily occur at the root.

To determine exactly what synchronization is required, it is sufficient to consider what dependences must be enforced. It is assumed that an entire innermost loop body (i.e., an entire iteration of the innermost loop) will be mapped to a single PE, so only loop-carried dependences are relevant. Since the problem class does not include conditionals, control dependences are not considered. The remaining dependences are loop-carried anti-, output, and flow dependences. Antidependences are enforced by ensuring that whenever a producer writes, any consumers needing the prior value have already completed. Output dependences are enforced by placing a total order on the producers writing to a particular location. Flow dependences are enforced by allowing the consumer to read a location only after the producer has placed a value in the location. In the basic method, all of these dependences were enforced through the placement of barriers between `doall` loops. An alternative is to use the state field in a more sophisticated manner. Identification information for the producer and a counter field decremented by each consumer can be placed into the state of each

[4]With constant dependence distances, this requires very little CPU time. If loop bounds and numbers of PEs are known statically, then almost all of this analysis can be performed at compile-time. Regardless, most of the analysis is static.

variable. Thus, the variable state becomes a tuple: (producer id, consumer count). The producer identification must be unique for each instance of a producer; it is sufficient to identify producers with an instance of the loop containing them (since only one producer writes to any shared location within a `doall`). For a series of `doall` loops, this identification can be the sequence number of the loop; for `doall`s within an enclosing loop, this information should also include values of outer loop induction variables.

The compile-time analysis for each array element written is in figure 26, and for each array element fetched is in figure 27. Note that the producer's write cannot proceed until the state of the location being written is zero (empty), and that the consumer's read cannot proceed until the state is positive. Also note that if loop bounds and the number PEs are known statically, the generated code can be optimized somewhat. Determining T is straightforward if dependence distances are constant. As in the basic method presented earlier in the paper, each PE executes exactly one tile. Thus, for convenience, the tile number referred to in the figures is equal to the number of the PE executing the tile.

The LHS will consist of an array reference of the form arr$[i_1, i_2, \ldots, i_d]$.

```
Find all sinks for this source by traversing each edge originating from this
vertex.

Using kill information from array dataflow analysis, remove edges to sinks which
are not reached from this source.

Generate code to determine the set of tiles T in which the sinks reside.

Let the producer's tile be m.  Generate code to perform the following:
     if T ≠ {m} then
          let k = the number of distinct tiles in which sinks reside = | T − {m} |
          write arr[i1, i2, ..., id] into commonLevel(T) with state set to (loop ID, k)
     fi
```

Figure 26: Analysis corresponding to the LHS of each assignment to an array element that is at the tail of a loop-carried dependence.

This method can be optimized by calculating the sharing required for each shared item only once, rather than once for the producer and then once for each consumer. Essentially, when the first producer or consumer of an item is reached in the parse tree, the dependence graph is traversed as in figure 26 for the producer or figure 27 for a consumer. Then the common memory unit can be stored in a look-up structure accessed by producer ID.

4.1 Examples

In section 3.3, an example with a pair of `doall` loop nests within an outer sequential loop is discussed. Applying the alternate method, communication can be added to the loop of figure 16 as shown in figure 28. For brevity and clarity, communication is shown only for the array `a`, and boundary conditions are omitted. Array `b` is handled similarly.

```
The RHS will include array references of the form arr[i_1, i_2, ..., i_d].

Find the source corresponding to this sink by traversing the edge terminating at
this vertex (array dataflow analysis info can be used to remove dead definitions,
so only one edge will enter each sink).

Find all sinks for the corresponding source by traversing each live edge
originating from the source vertex.

Let the producer's tile be t and the current sink's tile be m.  Generate code to
perform the following:
     if t ≠ m then
          determine T ∪ {t}, where T is the set of tiles in which the consumers
               reside
          when arr[i_1, i_2, ..., i_d].state = (producer ID, k) for k ≥ 1,
               fetch arr[i_1, i_2, ..., i_d] from commonLevel(T ∪ t), and decrement the
               associated state
     fi
```

Figure 27: Analysis corresponding to each RHS reference to an array element that is at the head of a loop-carried dependence.

In figure 28, PE_m first determines the limits of its tile exactly as in the matrix multiplication example of section 3.2. Then the loop is executed. After `a[i1, i2+1, j]` is calculated, the PE must determine whether the value is used elsewhere; to assist in this, we use a function `tile` which is passed a point in the iteration space, and returns the corresponding tile (or, equivalently, PE) number. If the value is used by another PE, it must be written to a location accessible by both PE_m and PE_t. This write can only proceed when when the location is empty (zero), and the write must set the location's state to one (the number of expected consumers) to indicate that a value has been produced. The fetch of `a[i1, i2, k+1]` is handled similarly.

As another example, consider figure 29, which contains a loop nest in which a single value is produced, but the value is referenced three times. After distributing the loop, this can be parallelized as shown in figure 30. Then communication can be added as specified in figure 31.

In figure 31 communication is placed following the definition of `a[i, j]` as specified in figure 26, and communication is placed before the references to `a[i-2, j-2]`, `a[i-4, j-4]`, and `a[i-8, j-8]` as specified in figure 27. Note that it is impossible to determine statically (without knowledge of the number of iterations and number of PEs) whether, in any given iteration, a particular `a[i, j]` will be referenced by one, two, or three PEs. Thus the conditionals are necessary.

4.2 A Modification to the Alternate Method

The alternate method can be modified slightly to place some of the reads and writes lower in the hierarchy. The modification is for the producer to determine at what levels data can be shared with each consumer. Then one write is performed for each level, with the state being set to the number of consumers reading at that level. The consumers must perform a complementary analysis.

PE$_m$, $0 \leq m < p$, executes the following:

```
Calculate lo²_m, hi²_m, lo³_m, hi³_m
for i1 := 1 to n1 do
      for i2 := lo²_m to hi²_m do
            for j := lo³_m to hi³_m do
                  a[i1, i2+1, j] := ...
                  t := tile(i1, i2+1, j-1)
                  if t ≠ m then
                        {a[i1, i2+1, j]@commonLevel(m, t); state.count=0; state.count:=1, state.id := m; store}
                  fi
            od
      od
      for i2 := lo²_m to hi²_m do
            for k := lo³_m to hi³_m do
                  t := tile(i1, i2-1, k+1)
                  if t ≠ m then
                        {a[i1, i2, k+1]@commonLevel(t,m); state.count> 0∧state.id= t;
                             state.count := state.count-1; fetch}
                  fi
                  ...:= a[i1, i2, k+1] ...
            od
      od
od
```

Figure 28: Loop of figure 16 with alternate communication generation.

This entails additional runtime analysis and a slightly larger number of writes performed by the producer. Since loops with constant dependence distances tend to have fairly small dependence distances[30] and a block partition is assumed, any given item is likely to be shared among at most a few PEs. Thus the cost of the additional writes is likely to be small, but the benefit of reduced latency reads may also be small. This method will be referred to as the *modified alternate method.*

Notice that, in the example of figure 16, the modified alternate method generates the same code as the alternate method; this code was shown in figure 28. The two methods differ in cases where a producer calculates a value used by more than one consumer; an example of this sort was shown for the alternate method in figure 29. For the modified alternate method, the generated code differs in two places: the code generated after a definition of shared data, and the code generated prior to a use of shared data. The code following the definition in the assignment of figure 30 is shown in figure 32. Note that most of the calculation can be done at compile time if the loop bounds and number of PEs are known statically. Complementary code is generated to precede uses of shared data.

4.3 Discussion

Three different methods for mapping programs to HSMAs have been presented. Each has its advantages and disadvantages, which can be summarized as follows:

```
for i := ...
      for j := ...
            a[i, j] := ...
            ...:= f(a[i-2, j-2], a[i-4, j-4], a[i-8, j-8])
      od
od
```

Figure 29: Sequential loop with single producer and multiple consumers.

```
doall i := ...
      doall j := ...
            a[i, j] := ...
      od
od
doall i := ...
      doall j := ...
            ...:= f(a[i-2, j-2], a[i-4, j-4], a[i-8, j-8])
      od
od
```

Figure 30: Loop of figure 29 after loop distribution, parallelization.

1. The basic method makes considerable use of regular sections. This makes the characterization of data sharing slightly inaccurate at times, and occasionally an item may be written multiple times to the same location in the hierarchy. Furthermore, a barrier is inserted between any two loops that arise following loop distribution. On the other hand, communication in the basic method is easily vectorized.

2. The alternate method entails exactly one write for each flow dependence crossing a tile boundary, and adds no barriers. This method is not as easily vectorized as the basic method, but communication traffic is likely to be more uniformly distributed in time.

3. The modified alternate method reduces access distances slightly over the second at the cost of more expensive runtime analysis and a slightly increased amount of overall traffic.

Which of these methods is most appropriate in any particular situation will be architecture and program-dependent. If the PEs support vectorized communication, the basic method may be the most attractive, especially if the section analysis is accurate and fast barrier synchronization hardware is available. The alternate method is likely to be more attractive if vector communication is not supported by hardware (e.g., for VLIW or superscalar PEs) and the dependence structure contains single producers writing values needed by multiple consumers, or if loop distribution results in loops with relatively little computation per tile (and thus between barriers in the first method). The modified alternate method may be chosen in cases described as attractive for the second, but in which there is a large variation in dependence distances and the tiles are small relative to the dependence distances.

PE$_m$, $0 \le m < p$, executes the following:

```
Calculate lo¹ₘ, hi¹ₘ, lo²ₘ, hi²ₘ
for i := lo¹ₘ to hi¹ₘ do
    for j := lo²ₘ to hi²ₘ
        a[i, j] := ...
        T := {tile(i-2, j-2), tile(i-4, j-4), tile(i-8, j-8) }
        if T ≠ {m} then
            k := | T − {m} |
            {a[i, j]@commonLevel(T); state.count=0;
                state.count:=k, state.id := m; store}
        fi
    od
od
for i := lo¹ₘ to hi¹ₘ do
    for j := lo²ₘ to hi²ₘ
        t := tile(i-2, j-2)
        if t ≠ m then
            Tt := {t, tile(i+2, j+2), tile(i+6, j+6)}
            {a[i-2, j-2]@commonLevel(Tt); state.count > 0 ∧ state.id = m;
                state.count := state.count-1; fetch}
        fi
        t := tile(i-4, j-4)
        if t ≠ m then
            Tt := {t, tile(i-2, j-2), tile(i+4, j+4)}
            {a[i-4, j-4]@commonLevel(Tt); state.count > 0 ∧ state.id = m;
                state.count := state.count-1; fetch}
        fi
        t := tile(i-8, j-8)
        if t ≠ m then
            Tt := {t, tile(i-6, j-6), tile(i-4, j-4)}
            {a[i-8, j-8]@commonLevel(Tt); state.count > 0 ∧ state.id = m;
                state.count := state.count-1; fetch}
        fi
        ...:= f(a[i-2, j-2], a[i-4, j-4], a[i-8, j-8])
    od
od
```

Figure 31: Loop of figure 30 with communication added.

```
t1 := tile(i+2, j+2)
t2 := tile(i+4, j+4)
t3 := tile(i+8, j+8)
if m ≠ {t1, t2, t3} then
      C := {commonLevel(m, t1), commonLevel(m, t2), commonLevel(m, t3)}
      ∀ c ∈ C:
            { a[i, j]@c; state.count = 0;
                  state.count := the number of consumers at commonLevel c, state.id := m; store}
```

Figure 32: Code following definition for modified alternate implementation of figure 30.

5 Related Research

Much research on parallelizing `for` loops has been performed. The effective use of local and hierarchical memory in this context has been recognized as an important problem. Among the approaches related to that of this paper are those of the Cedar, Kali, and FORTRAN D projects.

5.1 Cedar

In the Cedar restructuring compiler, two passes are used to optimize data placement for the hierarchical memory consisting of three levels: local, intermediate or cluster, and global[9, 32]. A globalization pass determines which data is shared among more than one cluster; this data is placed in the global memory. A privatization pass then determines which variables have a lifetime of just one loop iteration; these variables can be placed in local memory. Other data can be placed in cluster memory. Transformations which increase the locality (or decrease the degree of sharing) include scalar and array expansion.

This is somewhat similar to our approach, but data placement is more dynamic in our scheme. For example, if our scheme were adopted for Cedar, more data would move between global and cluster levels as a function of time than is currently the case. Such dynamic movement of data has been identified as a motivation for hierarchical shared memory architectures[15].

5.2 Kali and FORTRAN D

Kali[18] and FORTRAN D[10, 14] are parallel programming languages with explicit data distribution directives and provisions for specifying `forall` style parallelism. Kali is intended to be used for programming distributed-memory multiprocessors as if they had a shared memory, whereas FORTRAN D is intended to execute efficiently on both shared- and distributed-memory multiprocessors; both languages give the appearance of shared memory. There are several relevant differences between the approaches used in Kali and FORTRAN D and that herein:

- In Kali and FORTRAN D, the data space is partitioned among the PEs and iterations are assigned to PEs based upon whether specific data is present; this is the owner computes rule. In our approach, iterations are assigned to PEs, and data is moved wherever it is needed. We expect to get better load balancing than is seen in these languages (e.g., for the Gaussian elimination example in [18]), but at the cost of more complicated analysis and possibly more communication.

- In Kali, data needed by other PEs is sent, then loop iterations requiring only local data are executed, data sent by other PEs is received, and finally iterations needing nonlocal data are executed. This entails slightly more work for the compiler in partitioning the tiles into local and nonlocal portions, but will afford better latency tolerance than the approach herein. An adaptation of this method for shared memory would involve moving the fetches to as early a point as possible and then overlapping the read requests with computation to a greater degree than we currently do by executing the local iterations first, followed by the iterations requiring nonlocal data.
- Kali and FORTRAN D provide a wider variety of data distributions, which are chosen by the programmer.
- Data is redistributed in FORTRAN D via explicit directives given to the compiler. Our data movement is automatic, reducing the programmer's workload. As can be seen from the example in section 3.5, however, our basic method could sometimes result in extra runtime cost.
- Kali and FORTRAN D support runtime analysis for non-structured problems. Nothing in the basic or alternate methods precludes the use of such runtime analysis.

6 Conclusions and Further Research

In this paper methods for automatically parallelizing a class of nested loops for hierarchical shared-memory multiprocessor architectures are presented. The first method consists of parallelizing the loop nest, tiling the iteration space, determining the *use/def* sets of the various processing elements, determining intersections between PEs' *def* and *use* sets, and finally the insertion of communication and synchronization statements. The other two methods derive communication directly from the dependence graph.

A number of issues merit further investigation. Among these issues are:

- The compiling methods should be generalized to less restrictive classes of loops. For example, allowing a wider variety of array subscripts and supporting `while` loops would make the methods more general purpose. Automatic support for cyclic and block-cyclic tilings should be studied.
- It would be desirable to automatically exploit inter-procedural and inter-object parallelism in an HSMA.
- In addition to the patterns of partial sharing of variables supported by the compilation methods of this paper, partially-shared variables can be *split* and *fuse*d in time as their degree of sharing changes[15]. This splitting and fusing is a generalization of ascend/descend algorithms[28] and software combining[40, 31]. Compiler recognition of and support for splitting and fusing of partially-shared variables is desirable.
- Hierarchical distributed shared-memory architectures should be studied; it appears that such architectures simplify the task of the compiler and of the programmer with the potential for improved performance.

Acknowledgments: The authors would like to thank John Boyd, Himanshu Gupta, David Lutz, Loren Schwiebert, N. S. Sundar, and Yu-Chee Tseng for comments on an earlier draft of this paper.

References

[1] Zahira Ammarguellat and W. L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–295, June 1990.

[2] Arvind and Robert A. Iannucci. Two Fundamental Issues in Multiprocessing. Technical Report 226-6, Laboratory for Computer Science, MIT, May 1987.

[3] Vasanth Balasundaram. A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, June 1990.

[4] Vasanth Balasundaram and Ken Kennedy. A Technique for Summarizing Data Access and its Use in Parallelism Enhancing transformations. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 41–53, 1989.

[5] Eugene D. Brooks III. The Shared Memory Hypercube. *Parallel Computing*, 6:235–245, 1988.

[6] Tony F. Chan. Hierarchical Algorithms and Architectures for Parallel Scientific Computing. In *International Conference on Supercomputing*, pages 318–329, June 1990.

[7] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *IEEE Computer*, pages 39–47, June 1990.

[8] Raja Das, Ravi Ponnusamy, Joel Saltz, and Dimitri Mavriplis. Distributed Memory Compiler Methods for Irregular Problems—Data Copy Reuse and Runtime Partitioning. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, pages 185–219. Elsevier Science Publishers, 1992.

[9] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua. Restructuring FORTRAN Programs for Cedar. In *International Conference on Parallel Processing*, volume I, pages 57–66, August 1991.

[10] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D Language Specification. Technical Report COMP TR90-141, Department of Computer Science, Rice University, December 1990. Revised April, 1991.

[11] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh. Cedar—A Large Scale Multiprocessor. In *International Conference on Parallel Processing*, pages 524–529, 1983.

[12] Thomas Gross and Peter Steenkiste. Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler. *Software—Practice and Experience*, 20(2):133–155, Feb. 1990.

[13] Paul Havlak and Ken Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[14] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler support for machine-independent parallel programming in fortran d. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, pages 139–176. Elsevier Science Publishers, 1992.

[15] D. N. Jayasimha. Partially Shared Variables and Hierarchical Shared Memory Multiprocessor Architectures. In *11th Annual IEEE International Conference on Computers and Communications*, pages 63–71, April 1992.

[16] D. N. Jayasimha and J. D. Martens. Some Architectural and Compilation Issues in the Design of Hierarchical Shared Memory Multiprocessors. In *International Parallel Processing Symposium*, pages 567–572, March 1992.

[17] Harry F. Jordan. HEP Architecture, Programming, and Performance. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, pages 1–40. The MIT Press, Cambridge, MA, 1985.

[18] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[19] J. Konicek, T. Tilton, A. Veidenbaum, C. Q. Zhu, E. S. Davidson, R. Downing, M. Haney, M. Sharma, P. C. Yew, P. M. Farmwald, D. Kuck, D. Lavery, R. Lindsey, D. Pointer, J. Andrews, T. Beck, T. Murphy, S. Turner, and N. Warter. The Organization of the Cedar System. In *International Conference on Parallel Processing*, volume I, pages 49–56, August 1991.

[20] Donald C. Lindsay. Towards a Shared Memory Hypercube. Technical Report CMU-CS-88-190, Carnegie Mellon University, Department of Computer Science, November 1988.

[21] Stephen A. Mabbs and Kevin E. Forward. Optimizing the Communication Architecture of a Hierarchical Parallel Processor. In *International Conference on Parallel Processing*, volume I, pages 516–520, August 1990.

[22] I. O. Mahgoub and A. K. Elmagarmid. Performance Analysis of a Generalized Class of m-Level Hierarchical Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):129–138, March 1992.

[23] J. D. Martens and D. N. Jayasimha. A Tree-Structured Hierarchical Memory Multiprocessor. Technical Report 61, Department of Computer and Information Science, The Ohio State University, December 1989. Revised May 1990. An extended abstract appeared in *International Conference on Parallel Processing*, volume I, pages 561–562, August 1990.

[24] Henk Meijer and Selim G. Akl. Optimal Computation of Prefix Sums on a Binary Tree of Processors. *International Journal of Parallel Programming*, 16(2):127–136, 1987.

[25] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[26] John M. Mellor-Crummey and Michael L. Scott. Synchronization without Contention. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[27] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[28] Franco P. Preparata and Jean Vuillemin. The Cube Connected Cycles: a Versatile Network for Parallel Computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[29] J. Ramanujam and P. Sadayappan. Tiling of Iteration Spaces for Multicomputers. In *International Conference on Parallel Processing*, volume II, pages 179–186, August 1990.

[30] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An Empirical Study of FORTRAN Programs for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[31] Peiyi Tang and Pen-Chung Yew. Software Combining Algorithms for Distributing Hot-Spot Addressing. *Journal of Parallel and Distributed Computing*, 10, 1990.

[32] Peng Tu and David Padua. Array Privatization for Shared and Distributed Memory Machines. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, pages 64–67, September 1992. Published in the January, 1993 issue of ACM SIGPLAN Notices.

[33] Sizheng Wei and Saul Levy. Efficient Hierarchical Interconnection for Multiprocessor Systems. In *Supercomputing '92*, pages 708–717, 1992.

[34] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[35] Michael E. Wolf and Monica S. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[36] Michael Wolfe. Iteration Space Tiling for Memory Hierarchies. In *Parallel Processing for Scientific Computing*. SIAM, 1987. Published 1989.

[37] Michael Wolfe. More Iteration Space Tiling. In *Supercomputing '89*, pages 655–664, November 1989.

[38] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989.

[39] Michael Wolfe. The Tiny Loop Restructuring Research Tool. In *International Conference on Parallel Processing*, volume II, pages 46–53, August 1991.

[40] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing Hot-Spot Addressing in Large-Scale Multiprocessing. In *International Conference on Parallel Processing*, pages 51–58, August 1986.

[41] Chuan-Qi Zhu and Pen-Chung Yew. A Scheme to Enforce Data Dependence on Large Multiprocessor Systems. *IEEE Transactions on Software Engineering*, pages 726–739, June 1987.