Compiler Support for Array Distribution on NUMA Shared Memory Multiprocessors

Tarek S. Abdelrahman and Thomas N. Wong

Technical Report CSRI-331
July 1995

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Compiler Support for Array Distribution on NUMA Shared Memory Multiprocessors*

Tarek S. Abdelrahman and Thomas N. Wong
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

Management of program data to reduce false sharing and improve locality is critical for scaling performance on NUMA shared memory multiprocessors. We use HPF-like data decomposition directives to partition and place arrays in data-parallel applications on Hector, a shared-memory NUMA multiprocessor. We describe a compiler system for automating the partitioning and placement of arrays. The compiler uses a number of techniques that exploit Hector's shared memory architecture to efficiently implement distributed arrays. Experimental results from a prototype implementation demonstrate the effectiveness of these techniques, and demonstrate the magnitude of the performance improvement attainable when our compiler-based data management schemes are used instead of the operating system data management policies.

keywords: data distribution, locality management, cache management, parallelizing compilers, NUMA multiprocessors.

1 Introduction

Shared memory multiprocessors offer a familiar model of programming and better support for operating systems and programming environments. Bus-based shared memory multiprocessors are severely limited in their scalability to large numbers of processors. Hence, modern shared memory multiprocessors employ a memory structure in which shared memory is physically distributed, and a scalable point-to-point interconnection network is used to support distributed-shared memory. This structure makes memory access time dependent on the distance between

*This research is supported by grants from the Natural Sciences and Engineering Council of Canada and from the Information Technology Research Center of Ontario.

a processor and memory. Hence, these multiprocessors are known as Non-Uniform Memory Access (NUMA) multiprocessors. Examples of NUMA multiprocessors include the Stanford Dash [11], the T3D system from Cray [5], the Convex Exemplar SP-1000 [4], and Toronto's Hector multiprocessor [16].

The non-uniform nature of memory access in NUMA multiprocessors makes it necessary to enhance locality of reference in applications. Data must be carefully managed, i.e, partitioned, placed and possibly replicated across the physically distributed memory, to enable a processor to make most of its accesses to the local portion of the shared memory. Such data management tasks have been traditionally delegated to the operating system as part of its page management policies [8, 9, 15]. However, these policies are blind to application data access patterns and fail to enhance memory locality over a wide range of applications [1, 8]. Even with full knowledge of application data access patterns, operating system policies deal with application data in units of pages, which is too coarse of a grain to accommodate the fine-grain sharing that exists in parallel applications. This can result in loss of locality and false sharing. Recent experimental evidence on NUMA systems indicates that significant losses in performance may result from reliance on operating system policies for data management [1, 13].

In this paper we argue that compilers be used to manage data on NUMA multiprocessors and to enhance locality of reference in data-parallel applications. These applications use large data arrays typically accessed in regular patterns within loop nests. We propose and demonstrate that HPF-like directives can be used to partition arrays and place a data partition in the physical memory of the processor using the data the most. More specifically, the paper describes the design of a compiler system for the Hector Multiprocessor which uses data decompositions directives to translate an input data-parallel program with such directives and parallel loops into a Single Program Multiple Data (SPMD) program with explicit parallelism and explicitly distributed arrays. The compiler uses a number of array allocation techniques to translate array references in the input program into references to distributed arrays with minimal or no run-time overhead. Experimental results from a prototype implementation of the compiler demonstrate that significant improvements in performance can be achieved by using our compiler to manage program data instead of using operating system policies.

The remainder of this paper is organized as follows. In section 2, the Hector multiprocessor is briefly described. In section 3, the design of our compiler and the techniques it uses to support array distribution are presented in details. In section 4, experimental results obtained using the prototype implementation of the compiler are presented and discussed. In section 5, related work is reviewed. Finally, in section 6 conclusions and future research directions are presented.

2 The Hector multiprocessor

Hector [16] is a scalable shared memory multiprocessor which consists of a set of processing modules connected by a hierarchical ring interconnection network, as shown in Figure 1. Four processing modules are connected by a bus to form a *station*. Stations are connected by a local ring to form a *cluster*. Clusters are then connected by a central ring to form the system. Each processing module consists of a 20-Mhz Motorola 88000 processor with 16 Kbytes of fast cache and a 4-Mbyte local portion of shared memory. A processing module accesses non-local portions of shared memory over the ring interconnect. A processor accesses it local cache in one processor cycle. In the absence of contention, a processor loads a 16-byte cache line from its local part of the shared memory in 19 processor cycles; from an on-station memory in 29 processor cycles; from an on-ring memory in 37 processor cycles; and from an off-ring memory in 46 processor cycles. There is no hardware mechanism for maintaining cache consistency in Hector; rather consistency is maintained by the operating system.

Hurricane [15] is the operating system of Hector, and it provides application programmers with light weight processes to express and manage parallelism. The processes run in a single

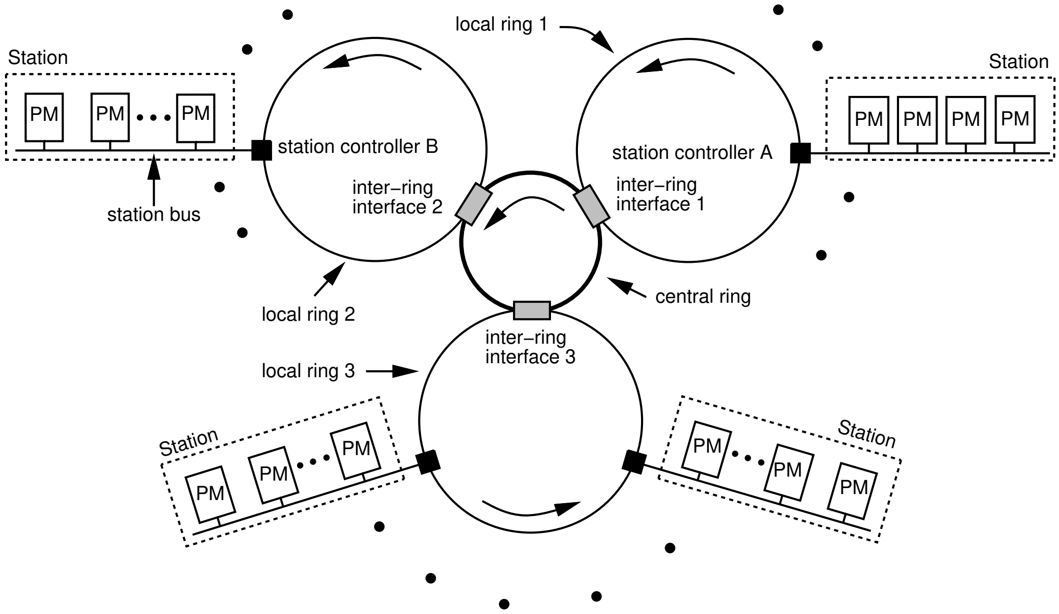


Figure 1: The Hector multiprocessor.

address space and share the same global variables, thus providing programmers with an easy mechanism to share and communicate data.

Hurricane plays a central role in data management on Hector. Since there are no mechanisms for cache consistency in hardware, the operating system maintains the data in the caches in a consistent state. The memory manager of Hurricane implements a single writer multiple reader (SWMR) consistency protocol for each 4-Kbyte page in memory. The manager keeps track of how pages are being accessed by the processors. A page is initially cacheable and remains in that state as long there is only a single writer to it, or as long are there are only multiple readers of it. Otherwise, the page becomes uncacheable, and can be accessed only from main memory. Once a page has become uncacheable, it remains as such until the programmer explicitly resets its state back to cacheable. The state-transition diagram of the SRMW protocol is depicted in Figure 2.

The operating system also controls the placement of data in the physically distributed shared memory via two page placement polices. The *First-hit* policy places a page in the

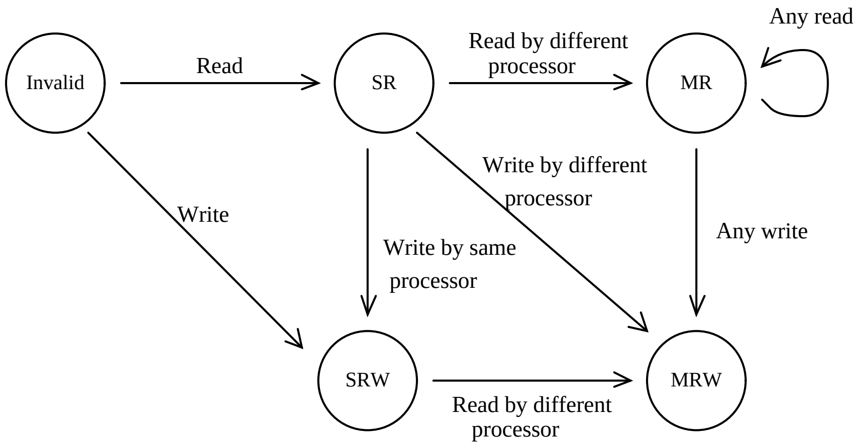


Figure 2: The SWMR Coherence Protocol.

physical memory of the first processor that generates a fault to the page. The *round-robin* policy places pages in the physical memories of the system in a round-robin fashion as the processors generate faults to pages. In both cases, once a page has been placed in a memory it remains there until the programmer explicitly resets its state, and then it can be re-placed using one of the above two policies. Pages are not replicated in memory.

3 Compiler design

3.1 Overview

The purpose of our compiler is to translate a data parallel program with data decomposition directives and parallel loops into a Single Program Multiple Data (SPMD) program with explicitly distributed arrays and with operating system constructs for the creation and synchronization of parallel tasks. The compiler consists of two main parts: a program analyzer and a program generator. The program analyzer computes the mapping of array partitions to processors, partitions parallel loops and classifies array references made by each processor as either local or non-local to the processor. The program generator produces code to allocate a local region in each physical memory for each array partition that resides in this memory, and produces code to create and synchronize light-weight processes that concurrently execute the iterations of the parallel loops.

3.2 Data decompositions and parallel loops

Data decomposition is achieved by specifying a partitioning scheme for each array in the program and by specifying a processor geometry to which array partitions map. This is done using the `DISTRIBUTE` and `PROCESSOR` directives respectively. The `PROCESSOR` directive declares an n-dimensional Cartesian grid of virtual processors $\mathbf{V}(V_0, V_1, \cdots, V_{n-1})$, where V_i is the number of processors in the i^{th} dimension of the grid, and $V_1 \times V_1 \times \cdots \times V_{n-1} = P$, the total number of processors. For example, `PROCESSOR P1(32)` declares a linear array `P1` of 32 virtual processors. Similarly, `PROCESSOR P2(4,6)` declares a 2-dimensional virtual processor grid `P2`, with 4 rows and 6 columns.

The `DISTRIBUTE` directive partitions an array by assigning a *partitioning attributes* to each dimension the array. The `DISTRIBUTE` directive also maps the resulting array partitions onto a processor geometry declared using the `PROCESSOR` directive. There are four partitioning attributes. The `BLOCK` attribute divides the corresponding dimension of the array in approximately equal size blocks such that a processor owns a contiguous range of that dimension of the array. The `CYCLIC` attribute divides the corresponding array dimension by distributing the array elements in this dimension to processors in a round-robin fashion. The `BLOCK_CYCLIC` attribute first groups array elements in the corresponding dimension in contiguous blocks of a given size, and then assigns the blocks to processors in a round-robin fashion. The block size, called the *block-cyclic factor*, is supplied by the programmer. Finally, the `NIL` attribute is used to indicate that the corresponding dimension of the array is not distributed. The application of the partitioning attributes to the dimensions of a multi-dimensional array results in useful data decomposition schemes. This is illustrated in Figure 3 using a 2-dimensional array `A`. The processor geometry on which the array is mapped determines the number of processors as-

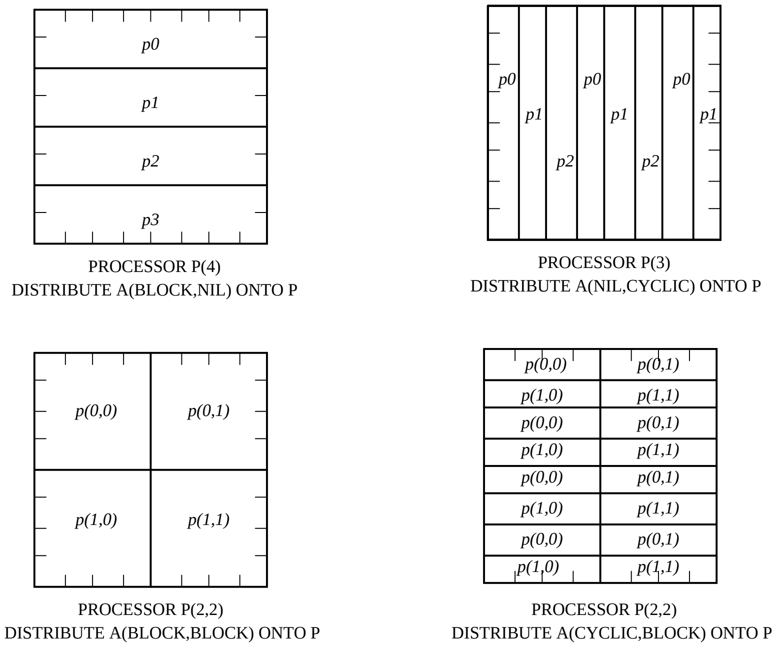


Figure 3: Array decomposition examples.

signed to each distributed dimension of the array. For example `DISTRIBUTE A(BLOCK,BLOCK) ONTO P`, where `P` is declared as `PROCESSOR P(2,4)` distributes the array `A` on 8 processors, assigning 2 processors to the first dimension of `A` and 4 processors to the second dimension.

Parallel loops are indicated in the input program using `FORALL` statements. A `FORALL` statement specifies a loop induction variable, a lower bound, an upper bound and a step size. The use of a `FORALL` statement implies that: (1) the iterations of the loop may be executed independently in parallel, and (2) all statements in a single iteration are executed sequentially without synchronization.

3.3 The program analyzer

The program analyzer uses the array decomposition directives specified in the input program to perform three main tasks: map array partitions to physical processors, partition parallel loops, i.e., determine the subset of loop iterations that will be executed by each processor, and analyze interprocessor communication to classify array references made by a processor as either local or non-local to the processor. The analysis performed by the program analyzer is largely based on that of Fortran D [6].

The mapping of data partitions to physical processors, is performed in two steps. In the first step, the array partitions are assigned to the virtual processors of the n-dimensional grid to which they are distributed. This is done using the `DISTRIBUTE` and `PROCESSOR` directives specified in the input program. In the second step, the virtual processors are assigned to the physical processors. The physical processors are viewed as a linear array, and are numbered $0, 1, \cdots, P-1$. Virtual processor $(v_0, v_1, \cdots, v_{n-1})$ is assigned to the physical processor numbered $\sum_{i=0}^{n-1} v_i \prod_{j=i+1}^{n-1} V_j$. Thus, for a 2-dimensional grid of virtual processors, this mapping scheme implies a column-major order assignment of virtual processors to physical processors. This mapping scheme does not take into consideration communication patterns in a parallel application, and how to best match them to the network topology on the target multiprocessor to minimize communication cost.

The program analyzer uses the owner-compute rule [6] to partition parallel loops. The owner-compute rule specifies that a processor executes a statement that *computes* an array element if and only if it *owns* the array element, i.e., has the element in its local memory. A processor executes a loop iteration if and only if the execution of this iteration causes the processor to write the array element. Consequently, the subset of loop iterations a processor executes is the subset of loop iterations that write local array elements. The partitioning of a parallel loop is performed by computing the *Local Index Set* for each distributed array on each processor. This is the set of array element indices that are locally owned by a processor. It is calculated using the array decomposition schemes provided by the programmer. The *Global Iteration Set* is computed for each array reference in the left hand side (`lhs`) of an assignment statement in the parallel loop. This set describes the iteration space spanned by the reference in the parallel loop, and is calculated using the inverse array subscript expressions appearing in the array reference. Finally, the *Local Iteration Set* is computed for the parallel loop on each processor. This set is the subset of loop iterations which are executed by each processor, and is computed by intersecting the Global Iteration Sets with the set that describes the loop iteration space.

The program analyzer also uses the owner-compute rule to determine which array references are non-local to each processor. Such references must be on the right hand side (`rhs`) of the statments of the body of the loop since the owner-compute rule makes all references on the `lhs` local. The *Local Access Set* for each `rhs` array reference in a loop statement is calculated by determining array indices spanned by the loop's Local Iteration Set. The *Remote Index Set* for each array reference is then computed as the set difference of the Local Access Set and the Local Index Set for the reference. Finally, the ownership of the array elements in the Remote Index Set is determined by intersecting the Remote Index Set with the Local Index Set for the array on each processor.

The analysis performed by the program analyzer is illustrated in Figure 4. It shows a simple example and the various index sets generated by the analyzer on a processor. The index sets are expressed using regular section descriptors, or RSD's [2]. An RSD is denoted by $[l_1 : u_1 : s_1][l_2 : u_2 : s_2] \cdots [l_n : u_n : s_n]$, where l_i, u_i and s_i are respectively the lower bound, upper bound and stride of the i^{th} dimension of the RSD.

3.4 The program generator

The program generator produces code to create light-weight processes, or threads, that execute the iterations of the parallel loops concurrently. The program generator replaces loop bounds in the input data parallel program by loop bound variables that are local to each processor. These variables are initialized to different values according to the physical processor number and to the Local Iteration Set of the loop, hence implementing loop partitioning. The program generator inserts code before each parallel loop to maintain the coherence of shared data (if necessary, see section 3.6) and inserts a barrier after each parallel loop.

```
PROCESSOR P(4,4);
DISTRIBUTE a (BLOCK,BLOCK) ONTO P;
DISTRIBUTE b (BLOCK,BLOCK) ONTO P;
    :
    :
FORALL (i, 0, 63, 1)      /* L1 */
   FORALL (j, 1, 62, 1)   /* L2 */
      a[i][j] = ( b[i][j-1] + b[i][j+1] ) / 2
```

(a) A data parallel code segment.

Local Index sets of a and b	=	$[32:47:1][16:31:1]$
Global Iteration Set of a	=	$[0:63:1][1:62:1]$
Local Iteration Set of $L1$	=	$[32:47:1]$
Local Iteration Set of $L2$	=	$[16:31:1]$
Local Access Set of b	=	$[32:47:1][15:30:1]\ +\ [32:47:1][17:32:1]$
Remote Access Set of b	=	$[32:47:1][15:15:1]\ +\ [32:47:1][32:32:1]$ $+\ [32:47:1][15:32:17]$
Remote ownership of b	=	$\{\ P(2,0):\ [32:47:1][15:15:1],$ $P(2,2):\ [32:47:1][32:32:1],$ others: [NIL] }

(b) Index sets for processor P(2,1).

Figure 4: Program analysis for a simple loop nest.

The program generator also produces code to allocate distributed arrays according to the data decomposition directives provided in the input program. This code allocates a region in the local memory of each processor to hold each array partition that resides in this local memory. This allocation results in an array which is physically distributed in shared memory, but is accessed in the output SPMD program as if it is allocated in contiguous form. Array references in the data parallel program are converted into equivalent array references to the distributed array in the SPMD program. This conversion is referred to as *reference translation.* The program generator takes advantage of the single address space and uses special array allocation schemes to perform reference translation without run-time overhead.

3.5 Reference translation

In this section we describe the techniques used to perform reference translation. These techniques are presented using 2-dimensional arrays, and assuming row-major storage order of arrays in memory. The techniques are equally applicable to higher dimension arrays and to column-major storage order.

Reference translation for an array is performed using a number of allocation schemes for the partitions of the array and using one or more array *descriptors*. A descriptor is an array of pointers in which each pointer points to a region of an array partition. The pointers are initialized by the program generator in such a way to allow the array to be accessed as if it is contiguous, when it is physically distributed in memory. The initialization of the pointers in a descriptor is referred to as *region binding*. Array references in the `lhs` of statements are always writes to local array elements, while array references in the `rhs` of statements are reads that may access local or non-local array elements. The translation of `lhs` array references is called *local reference translation*; the translation of `rhs` array references is called *global reference translation*.

3.5.1 Local reference translation

Local reference translation is performed using *local descriptors*. There is one local descriptor for each distributed array in each processor; the local descriptor of an array is a private variable to a processor and is used only to access the processor's own portion of the array. The binding of each local descriptor for an array depends on the partitioning attribute assigned the last dimension of the array. If the attribute assigned to the last dimension of the array is `NIL`, a contiguous local region is allocated in the memory of each processor to hold the rows of each local partition. The binding of the local descriptor is done by assigning the addresses of the rows of the partition to corresponding rows of the local descriptor. This is shown in Figures 5 and 6 for the `(BLOCK,NIL)` and `(CYCLIC,NIL)` decompositions respectively.

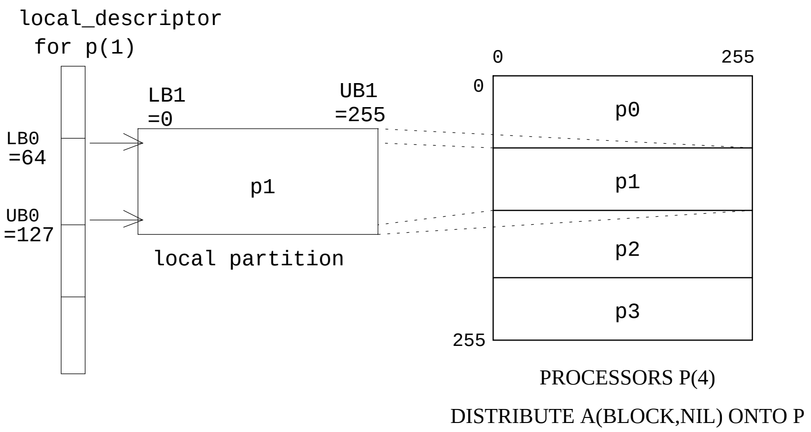


Figure 5: Region binding for (BLOCK,NIL) decomposition.

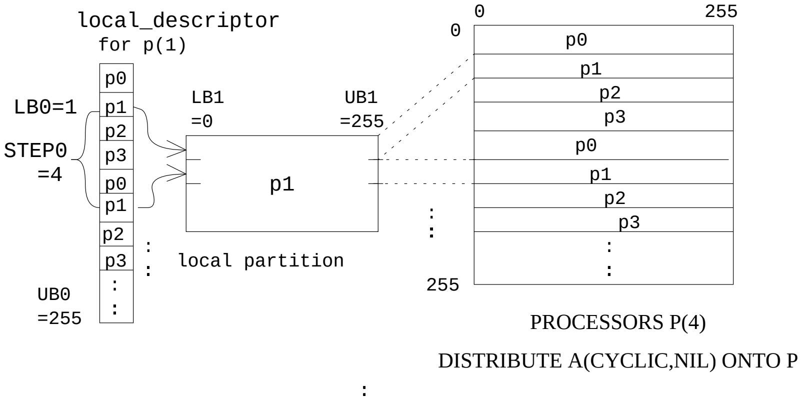


Figure 6: Region binding for (CYCLIC,NIL) decomposition.

If the partitioning attribute assigned to the last dimension of an array is BLOCK, a contiguous region of memory is allocated to hold the row segments of the local array partition. Region binding is done by assigning the address of each row of the array partition to the corresponding row in the local descriptor, with an offset adjustment. The offset added to each row pointer corresponds to the starting position of left column boundary of the local array partition. This is illustrated for the (NIL,BLOCK) and (BLOCK,BLOCK) decompositions in Figures 7 and 8 respectively.

If the partitioning attribute assigned to the last dimension of any array is CYCLIC, the whole row is allocated and bound to the local descriptor. The processor uses only a subset of

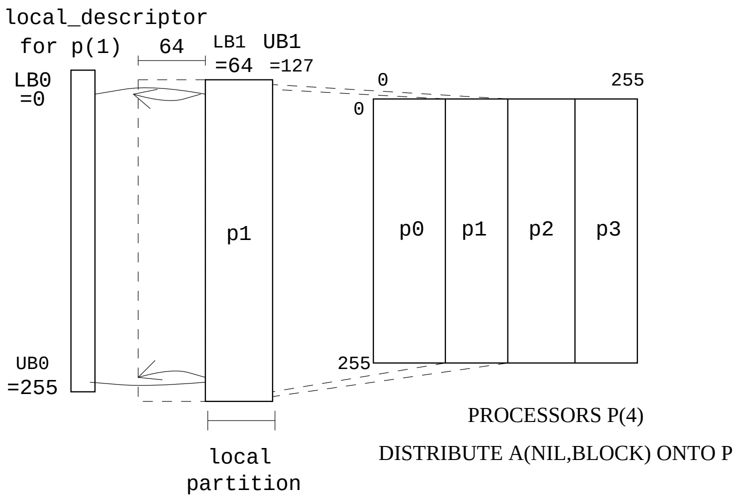


Figure 7: Region binding for (NIL,BLOCK) decomposition.

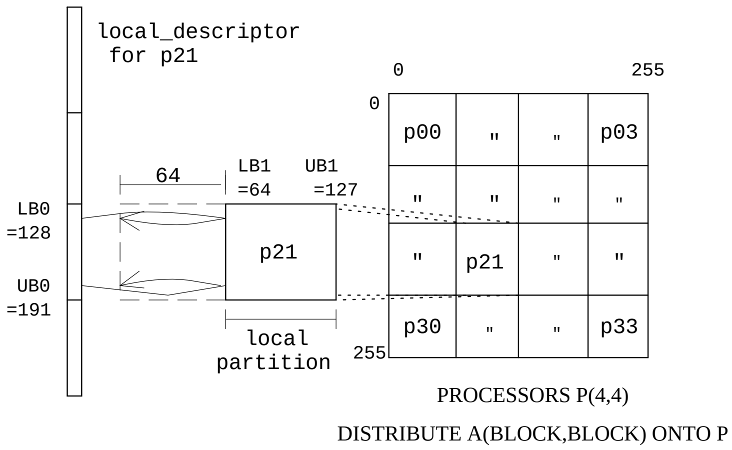


Figure 8: Region binding for (BLOCK,BLOCK) decomposition.

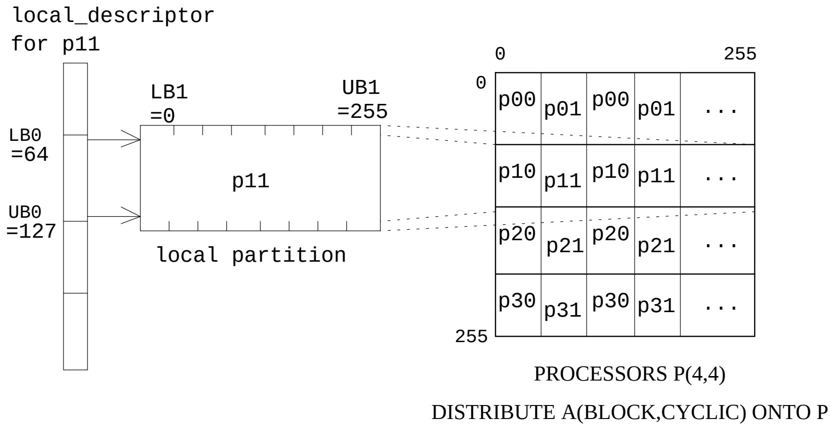


Figure 9: Region binding for `(BLOCK,CYCLIC)` decomposition.

this region, which corresponds to the elements it owns; the remaining elements are not used. Hence, this allocation scheme results in memory overhead. Compared to the previous two cases in which only the amount of storage needed by a processor is allocated, this scheme for region binding is less space-efficient. In section 3.7, a number of optimizations will be presented to eliminate this memory overhead. Region binding for the `(BLOCK,CYCLIC)` decomposition is shown in Figure 9.

3.5.2 Global reference translation

Global reference translation is used to translate references to arrays in the `rhs` of statements, which are read references to arrays elements that are both local and non-local to a processor. *Global descriptors* are used to perform global address translation. Unlike local descriptors, global descriptors are shared variables; there is only one global descriptor for each distributed array, and all processors share this descriptor.

There are two methods for binding the global descriptor of any array to the various partitions of the distributed array. *Remote region binding* is used when the `NIL` attribute is assigned to the last dimension of an array. In this case, each row pointer is assigned to the corresponding row's location in the global descriptor. This is depicted in Figure 10 for `(BLOCK,NIL)` decom-

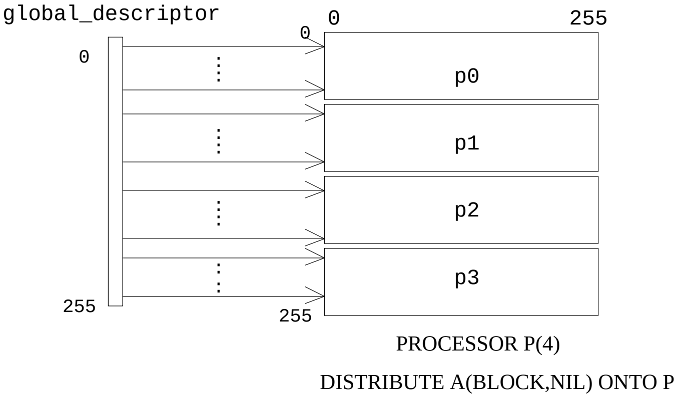


Figure 10: Remote region binding for `(BLOCK,NIL)` decomposition.

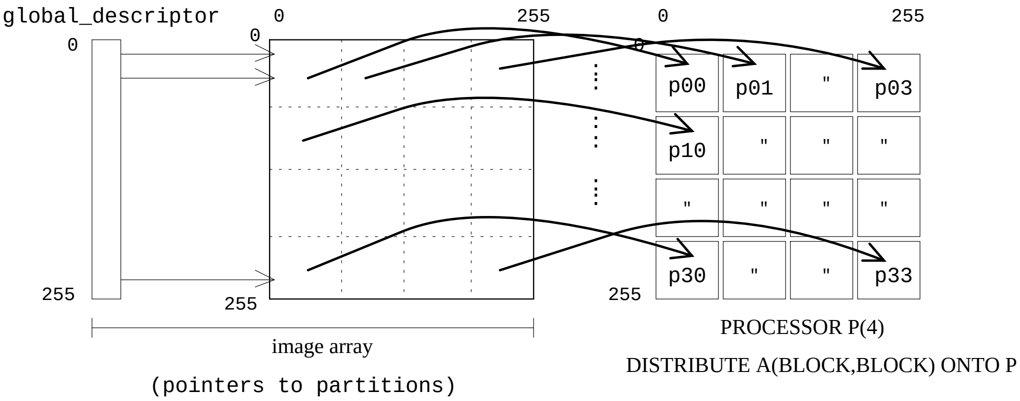


Figure 11: Image binding for `(BLOCK,BLOCK)` decomposition.

position. *Image binding* is used when the partitioning attribute assigned to the last dimension of the array is not `NIL`. An *image array* is an array of pointers of the same size and dimensions as the data array. Each element in image array is a pointer to the corresponding element of the distributed array. The image array is used to access the elements of the distributed array through pointer dereferencing. The use of image binding is depicted in Figure 11 for `(BLOCK,BLOCK)` decomposition.

The use of the image array to perform global reference translation results in memory overhead since an additional image array is used for each data array. In section 3.7 a number of optimizations to eliminate this overhead will be described.

3.6 Shadow regions

In this section, we describe a scheme to reduce false sharing in distributed arrays. This scheme is orthogonal to the data distribution schemes, in that it can be used with any data decomposition scheme.

In this scheme, the compiler uses *shadow regions* to maintain cache consistency. Shadow regions are local memory storage used to hold non-local data in the local memory of a processor during the execution of a parallel loop. A processor accesses non-local data by accessing the local shadow region. The program generator inserts code to maintain memory consistency of data between local shadow regions and non-local array partitions.

Program analysis is used to determine interprocessor communication for each distributed array on each processor. This allows the compiler to determine the size and location of local buffers required to hold the non-local data. The compiler allocates a memory region in each processors that is large enough to hold both the local array partition as well as the shadow region. The compiler generates code such that the remote data is accessed correctly in the program by copying data from data partition into the shadow region before each parallel loop.

Once the shadow region is created, it is bound to a local descriptor variable. This binding process is called *shadow region binding*. The process of binding a shadow region is similar to that of binding a local partition. Figure 12 depicts shadow region binding for `(BLOCK,BLOCK)` decomposition.

The compiler generates prefetching code to copy non-local data into the shadow region before each parallel loop. This code employs a two-step copying process that leaves both local and non-local data partitions cacheable. The processor containing the non-local data first copies the data into a temporary buffer. The processor accessing the non-local data then copies the temporary buffer into the shadow region, which leaves only the buffer uncacheable. This copying process also maintains data coherence between the non-local memory and the local shadow memory through the uncacheability of the buffer. The non-local processor invalidates

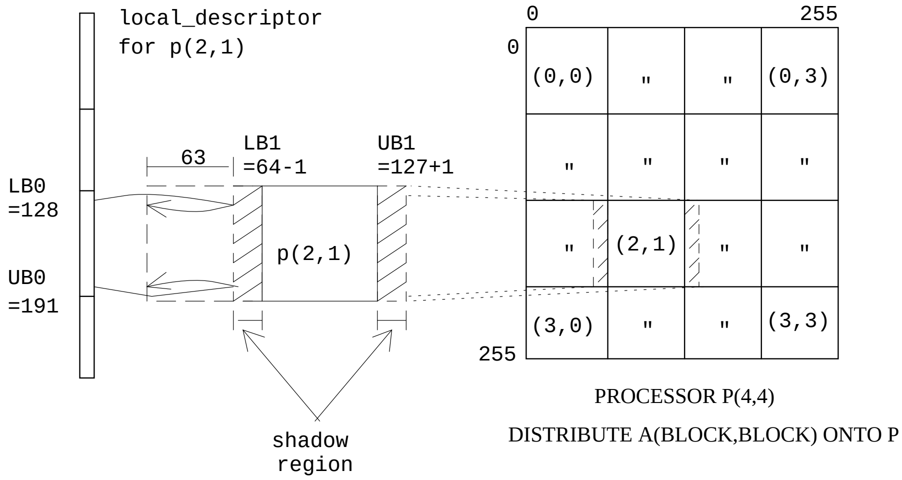


Figure 12: Shadow region binding for (BLOCK,BLOCK) decomposition.

the appropriate cache lines out of its cache as it writes the data to the temporary buffer and the local processor reads the data from memory.

The use of shadow regions has two main advantages. First, a shadow region provides faster access to non-local data. Array references in the `rhs` of a loop statement are translated into local shadow region accesses. A non-local array element is copied once into the local shadow region of a processor, and is then read by the processor from the shadow region. Hence, when there is reuse of non-local array data in a parallel loop, the cost of non-local memory access is incurred only once rather than every time the non-local element is accessed. Second, since the shadow and the local partition are merged in one local region, there is no longer a distinction between local and non-local accesses, which simplifies generated parallel code by avoiding the possible use of image arrays.

However, shadow regions result in memory overhead since they are implemented as extensions to local partitions. This overhead is small in parallel applications with high spatial locality in which a processor accesses "border" data from neighboring processors. The overhead can be large in applications which exhibit little spatial locality making non-local accesses sparsely distributed over the array; shadow region elements which are not used to store data used by the processor are wasted.

3.7 Reduction of memory overhead

Memory overhead results when the CYCLIC attribute is assigned to the last dimension of an array, and hence, more storage is used than what is needed to store the local partition of an array in a processor's local memory. Memory overhead also results when the image array is used in global reference translation. Memory overhead not only increases the amount of memory used by a program, but also degrades program performance. The use of the image array increases data access time because of pointer dereferencing. Also, image array elements can eject data array elements from the local cache either due to limited cache capacity or due to conflicts, which increases the number of cache misses incurred by the program.

In this section, we present a number of optimizations to reduce memory overhead by eliminating the need for the imaging array. These optimizations are: dimension swapping, dimension splitting and loop splitting. We also describe an index transformation to reduce memory overhead associated with CYCLIC distributions.

3.7.1 Dimension swapping

Dimension swapping is an optimization that allows global reference translation to be performed using remote region binding instead of image binding, hence, eliminating the overhead of the image array altogether. Consider an n-dimensional array $A[s_0][s_1]\ldots[s_{n-1}]$ with the partitioning scheme $P(p_0, p_1, \ldots, p_{n-1})$, where $s_0, \ldots, s_{n-1}$ are the sizes of the respective dimensions of A, and $p_0, \ldots, p_{n-1}$ are the partitioning attributes of the respective dimensions of A. Dimension swapping can be used when the last partitioning attribute P_{n-1} is not NIL and one of the other attributes is NIL. Let the dimension to which the NIL attribute is assigned be p_d; that is, $p_d =$ NIL, $\quad 0 \leq d \leq n-2$. The array A is transformed into an array $A'[s'_0][s'_1]\ldots[s'_d]\ldots[s'_{n-2}][s'_{n-1}]$ with the new partitioning scheme $P'(p'_0, p'_1, \ldots, p'_d, \ldots, p'_{n-2}, p'_{n-1})$ such that

$$
\begin{aligned}
s'_i &= s_i; \qquad 0 \leq i \leq n-2, i \neq d \\
s'_d &= s_{n-1};
\end{aligned}
$$

$$
\begin{aligned}
s'_{n-1} &= s_d; \\[1em]
p'_i &= p_i; \qquad 0 \le i \le n-2, i \ne d \\
p'_d &= p_{n-1}; \\
p'_{n-1} &= p_d.
\end{aligned}
$$

That is, the new array A' is expressed as $A'[s_0][s_1]\ldots[s_{n-1}]\ldots[s_{n-2}][s_d]$, and has the partitioning scheme $P'(p_0, p_1, \ldots, p_{n-1}, \ldots, p_{n-2}, p_d)$. In the program, a reference to the array A is replaced by a reference to A' in which the subscript expressions in dimensions d and $n-1$ are swapped. For example, the reference $A(i_0, i_1, \ldots, i_d, \ldots, i_{n-2}, i_{n-1})$ is replaced by the reference $A'(i_0, i_1, \ldots, i_{n-1}, \ldots, i_{n-2}, i_d)$.

Dimension swapping improves performance by eliminating the the overhead of the image array. However, it may lead non-unit stride accesses of the array, which can degrade cache performance. The extent to which this optimization improves performance is application dependent.

3.7.2 Dimension splitting

Dimension splitting also allows the use of remote region binding instead of image binding in global reference translation. It is used when the partitioning attribute of the last dimension of the array is not `NIL`, and there is no `NIL` attribute assigned to any other dimension of the array, hence, preventing dimension swapping.

Consider an n-dimensional array $A[s_0][s_1]\ldots[s_{n-2}][s_{n-1}]$ whose partitioning scheme is given by $P(p_0, p_1, \ldots, p_{n-2}, p_{n-1})$, and in which the last partitioning attribute p_{n-1} is not `NIL`. The array n-dimensional array A is transformed into an equivalent $(n+1)$-dimensional array $A'[s_0][s_1]\ldots[s_{n-2}][s'_{n-1}][s'_n]$, where $s_{n-1} = s'_{n-1} \times s'_n$. That is, the last dimension size s_{n-1} in A is split into two dimensions of sizes s'_{n-1} and s'_n in A'. The splitting is such that the combined size $s'_{n-1} \times s'_n$ in A' is equivalent to the original size s_{n-1} in A. For the sake of presentation, we assume that the size of the last dimension s_{n-1} is divisible by V_{n-1}, where

V_{n-1} is the number of processors used in the last dimension of the data distribution of A. This divisibility restriction is not necessary in general.

The new array A' is partitioned using the partitioning scheme $P'(p_0, p_1, \ldots, p_{n-2}, p_{n-1}, p'_n)$, where $p'_n = \texttt{NIL}$. To achieve this, we divide the last dimension size s_{n-1} into s'_{n-1} and s'_n such that $s'_{n-1} = V_{n-1}$ and $s'_n = s_{n-1}$ div V_{n-1}. In the program, a reference $A(i_0, \ldots, i_{n-2}, i_{n-1})$ is replaced by the equivalent reference $A'(i_0, \ldots, i_{n-2}, i'_{n-1}, i'_n)$. The last two subscript expressions i'_{n-1} and i'_n in A' can be expressed in terms of the two subscript expressions i_{n-1} and i_n as follows. If p_{n-1} is `BLOCK`, then $i'_{n-1} = i_{n-1}$ div B, and $i'_n = i_{n-1}$, where $B = s_{n-1}$ div V_{n-1}. If p_{n-1} is `CYCLIC`, then $i'_{n-1} = i_{n-1} \bmod V_{n-1}$, and $i'_n = i_{n-1}$.

The array A' is a distributed array with the last partitioning attribute $p_{n-1} = \texttt{NIL}$. This allows the use of local region binding with offsets that can be reflected statically in the local descriptor of the array.

Dimension splitting allows the use of the more efficient remote region binding instead of image binding. However, it introduces run-time overhead in array indexing due to the use of integer division and modulus operators. The extent to which the optimization improves performance will depend on the cost of these operators relative to the cost of using the image array, which is dependent on both the application and on the target processor architecture.

3.7.3 Loop splitting

Loop splitting is an alternative optimization for replacing image array binding with remote region binding. It is used when dimension swapping is not possible. In this optimization, loop iterations that cause `rhs` array references in a statement to read non-local data are peeled off the loop [18]. The peeling process transforms each loops nest into an equivalent sequence of loop nests such that in each of the resulting loop nest, `rhs` array references are to one and only one processor. This allows the compiler to replace all references to the array by a reference to the corresponding region of the array.

Algorithm LoopPeel
Input: peeled iteration set $\mathcal{L}$
a set of `rhs` array references $\mathcal{R}$
a list of processor locations $\mathcal{P}$ = `NIL`
Output: a set of tuples $\mathcal{S} = \{(\mathcal{X}, \mathcal{M})\}$
where $\mathcal{X} \in \mathcal{L}$ and $\mathcal{M}$ denotes the processor locations

begin
  let R = first reference in $\mathcal{R}$
  for each processor location p for R
    find the iteration set $\mathcal{X} \in \mathcal{L}$ such that R causes a reference to the processor p
    if $\mathcal{X} \neq$ `NIL`
      let $\mathcal{M} = \mathcal{P}$
      add p to $\mathcal{M}$
      if $\mid \mathcal{R} \mid == 1$
        add $(\mathcal{X}, \mathcal{M})$ to the set $\mathcal{S}$
      else /* $\mid \mathcal{R} \mid > 1$ */
        let $\mathcal{R}' = \mathcal{R} - R$
        call LoopPeel($\mathcal{X}$, $\mathcal{R}'$, $\mathcal{M}$)
end

Figure 13: The Algorithm for Loop Splitting.

The iterations that need to be peeled off a loop can be easily determined using the inverse of `rhs` array subscript expressions in the loop. However, it is also necessary to determine the ownership of the remote elements accessed in each iteration by a given reference to translate this reference. The algorithm shown in Figure 13 is used for this purpose. The algorithm takes the set of loop iterations that must be peeled, the set of `rhs` array references and determines for each array reference which loop iterations cause a reference on each processor. This information is then used to split the loops as described above.

Loop splitting improves performance by eliminating the need for image array binding. However, it degrades performance because of increased loop overhead. The extent to which it benefits performance will depend on loop overhead cost relative to image array cost. The extent to which is improves performance relative to the dimension splitting optimization will depend on the cost of the integer divide and modulus operator relative to the introduced loop overhead.

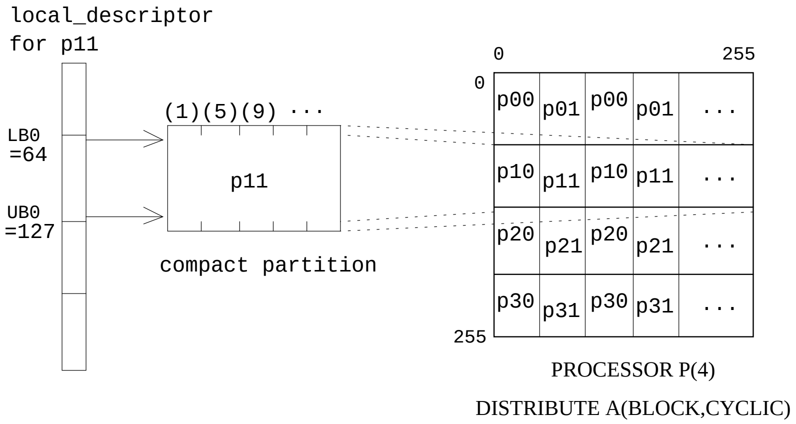


Figure 14: Compact region binding for `(BLOCK,CYCLIC)` decomposition.

Since the number of iterations peeled off a loop depends on the communication patterns in the application, the relative benefit of this optimization is also application dependent.

3.7.4 Region Compaction

Region compaction eliminates memory overhead that occurs when the `CYCLIC` attribute appears in the last dimension of an array. Instead of allocating the whole row for each row in the local regions, only an amount of memory is allocated to store the elements the processor owns. Local regions allocated in this way will be referred to as *compact regions*. Figure 14 shows compact region binding for the `(BLOCK,CYCLIC)` decomposition.

A reference to an array through the local descriptor requires that array index expressions be translated. A $A(i_0, \ldots, i_{n-2}, i_{n-1})$, is replaced by the reference $A'(i_0, \ldots, i_{n-2}, i_{n-1} \text{ div } V_{n-1})$, where A' uses the local descriptor for A and V_{n-1} is the total number of processors used in the last dimension of processor grid on which A is mapped.

This optimization eliminates memory overhead, but increases the amount of computation needed to perform array indexing. The extent to which it improves performance depends on the cost of the div operator relative to the cost of using the image array. This is dependent on both the application and on the target processor.

The memory overhead associated with having a `CYCLIC` attribute in the last dimension of an array can also be eliminated using dimension swapping. In this case, the objective is to swap a non-CYCLIC attribute into the last dimension of the array.

4 Experimental results

A prototype of the compiler described above has been implemented for the Hector Multiprocessor and applied to a number of application kernels. The implementation takes as input programs written in a restricted version of the C programming language [17], and produces C code with Hurricane system calls to create and synchronize parallel threads. In this section, we describe the experimental results obtained from a 16-processor cluster consisting of 4 stations on a single local ring. The results demonstrate the performance improvements that can be obtained by using our compiler on the application benchmarks. These improvements compare favorably to the improvements obtained using the operating system data management policies, and to the improvements obtained using a set of data management macros called Numacros (described in the next section). The results also show the impact of memory reduction optimizations on performance.

The four application kernels used in our experiments are: Jacobi, Matrix Multiply, LU factorization, and Simplex.

Jacobi. This kernel implements a successive-over-relaxation algorithm for iteratively solving partial differential equations. The main part of the program is shown below.

```
repeat
   FORALL i=2,n-1
       FORALL j=2,n-1
           b[i][j]=(a[i-1][j]+a[i+1][j]+
               a[i][j-1]+a[i][j+1])/4.0
   FORALL i=2,n-1
       FORALL j=2,n-1
           a[i][j]=b[i][j]
until convergence
```

Matrix Multiply. This kernel implements a standard matrix multiplication algorithm which computes the product of two square matrices A and B as follows:

```
FORALL i=1,n
    for k=1,n
        r = A[i][k]  % register allocated
        for j=1,n
            C[i][j] += r*B[k][j]
```

LU factorization. This kernel decomposes an $\mathtt{n} \times \mathtt{n}$ square matrix a into a lower and an upper triangular matrices using Gauss-Jordan elimination without pivoting. The main part of the kernel is as follows:

```
for k=1,n-1
    % find index of pivot
    ipivot =index(max(a[k][k],n-k))
    % exchange pivot and current rows
    exchange(k,ipivot)
    FORALL i=k+1,n
        % compute multipliers
        m[i]= -a[i][k]/a[k][k]
    % perform Gauss-Jordan elimination
    for i=k+1,n
        for j=k+1,n
            a[i][j]=a[i][j]+m[i]*a[k][j]
```

Simplex. This application program solves an $\mathtt{m} \times \mathtt{n}$ linear programming problem using the well-known simplex method. The method uses an $(\mathtt{m}+1) \times (\mathtt{n}+1)$ matrix t, referred to as the simplex tableau, and it iterates over that tableau until a solution is found. The details of the method can be found in [14]. The main part of each iteration consists of:

```
% find pivot column
pc = index(min(t[m+1][*]))
% find pivot row
pr = index(min(t[*][n+1]/t[*][pc]))
% perform Gauss-Jordan elimination
FORALL i=1,m+1
  FORALL j=1,n+1
    if i != pr
      t[i][j] = t[i][j] -
                (t[i][pc]/t[pr][pc])*t[pr][j]
    else
      t[pr][j]=t[pr][j]/t[pr][pc]
```

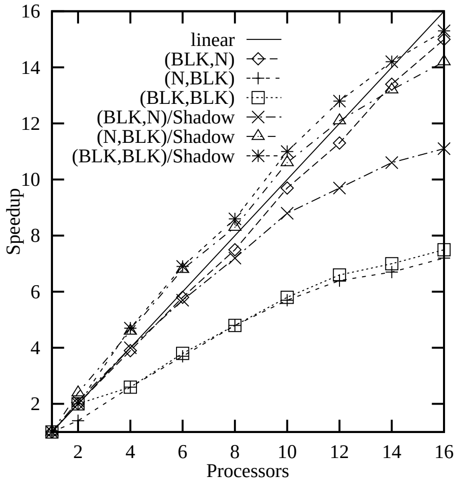
4.1 Results and Discussion

The performance of the compiler-generated parallel kernels on Hector is shown in Figure 15. The performance is measured by the speedup, which is defined as the ratio of the execution time of the sequential kernel to the execution time of the compiler-generated parallel kernel. The data sizes for each kernel used to measure performance is selected so as all data used by the sequential kernel fits in the local memory of one processor module. This is done to ensure that speedup numbers are not inflated due to the increase in memory locality when more than one processor is used; using a large data set would make the sequential program slower due to the need for non-local memory allocation, and consequently non-local memory accesses.

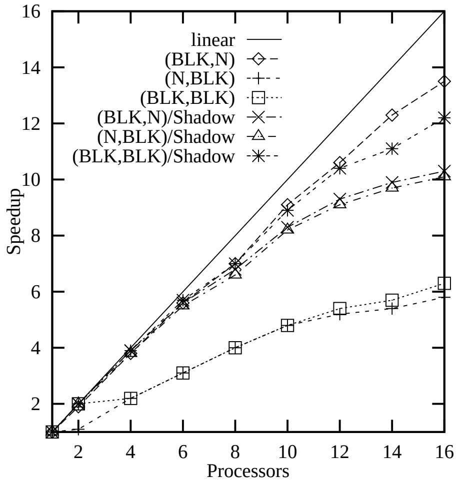
The results generally indicate that the speedup of the compiler-generated parallel codes is relatively scaling with the number of processors. The extent to which the performance of each kernel improves, and the data decomposition scheme needed to achieve such improvement varies from one application to another, and depends on the number of processors.

The performance of Jacobi is shown for three input data sizes (i.e., sizes of arrays `a` and `b`), 350×700, 400×400 and 700×350. The performance of the kernel without the use of shadow regions scales well for the `(BLOCK,NIL)` scheme, but is poor for the `(NIL,BLOCK)` and `(BLOCK,BLOCK)` schemes. The poor performance for these two schemes is attributed to the resulting pattern of remote data accesses, which causes uncacheability of data. Since arrays are stored in row-major order, accessing the non-local data in column-major order causes most, if not all, of the pages in non-local array partitions to become uncacheable. In contrast, the (BLOCK,NIL) scheme makes uncacheable only pages containing boundary rows in the partitions, which does not impact on performance in a significant way. In addition, the `(NIL,BLOCK)` and `(BLOCK,BLOCK)` schemes require the use of an image array for remote reference translation, which further degrades performance.

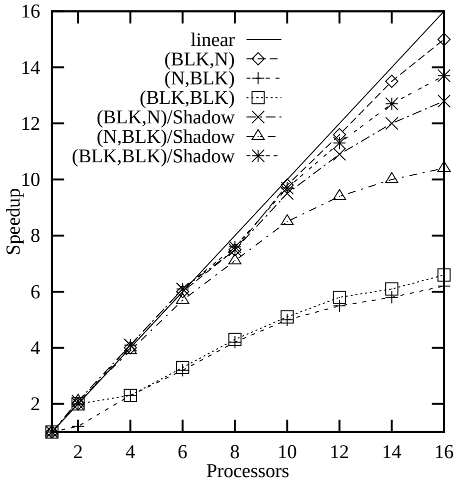
The performance of Jacobi with the use of shadow regions is generally scaling for all data decomposition scheme. The performance is largely determined by the amount of remote data



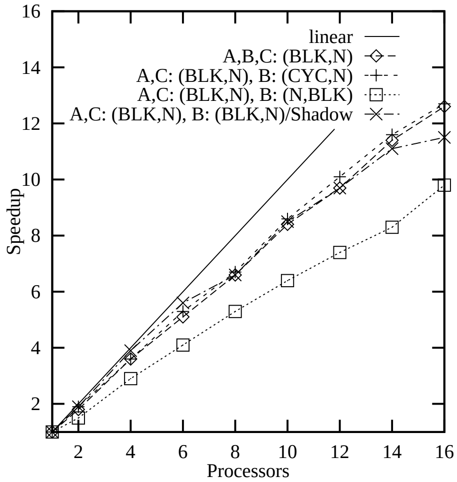
(a) Jacobi (350 × 700)



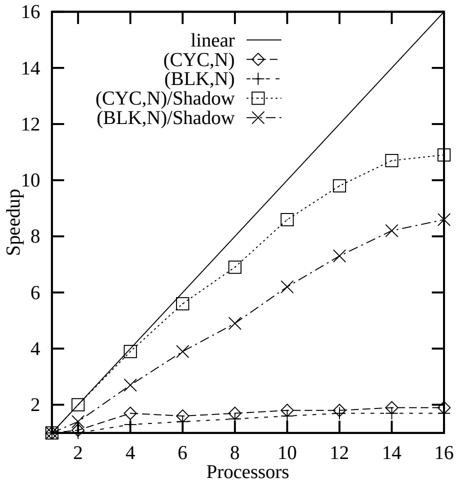
(b) Jacobi (400 × 400)



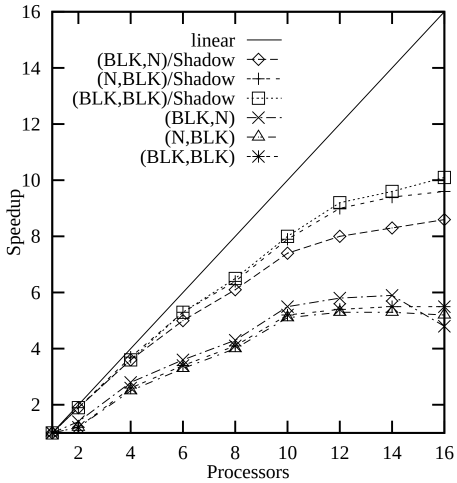
(c) Jacobi (700 × 350)



(d) Matrix Multiply (400 × 400)



(e) LU Factorization (512 × 512)



(f) Simplex (257 × 513)

Figure 15: Performance of compiler generated parallel programs.

communication in the parallel program. In the case of Jacobi, this is determined by the number of data elements shared between pairs of processors. In the case of the `(NIL,BLOCK)/Shadow` scheme, boundary columns in the arrays `a` and `b` are shared, and the scheme performs better than the `(BLOCK,NIL)/Shadow` scheme in the case where the column size is smaller than the row size of the array; i.e. 350×700. Similarly, `(BLOCK,NIL)/Shadow` scheme performs better than the `(NIL,BLOCK)/Shadow` scheme when the row size is smaller than the column size, i.e., 700×350. As expected, the `(BLOCK,BLOCK)/Shadow` scheme out-performs both in for all data sizes since it always minimizes communication volume.

The performance of the Jacobi kernel with the use of shadow regions is better than its performance without shadow regions when false sharing is a major factor affecting performance. The implementation of shadow regions overcomes false sharing, resulting in better performance. However, when false sharing is not a major factor, the performance of the kernel without the use of shadow regions is better that its performance with shadow regions. This is due to the overhead associated with data copying in the case of shadow regions.

The superlinear behavior exhibited by the Jacobi kernel for the data size of 350×700 is due to the decrease in the size of the data handled per processor as larger number of processors are used, and the increase opportunity for data re-use offered by the column and grid block distributions. The decreasing size of the local data makes it more likely for the data to fit in the local cache.

The performance of the matrix multiply kernel is shown in Figure 15(d). A `(BLOCK,NIL)` distribution is used for arrays `A` and `C` since it maximizes the locality of these arrays. The array `B`, which is accessed in its entirety by all processors is distributed using a number of schemes. The figure indicates that the schemes, with the exception of `(NIL,BLOCK)` result in good performance. The poor performance in the case of the `(NIL,BLOCK)` scheme is due to the need to use the image array to perform global reference translation. The `(NIL,CYCLIC)` distribution results in similar performance, and is not shown.

The performance of the matrix multiply kernel for the remaining schemes is fairly comparable. The shadow scheme performs slightly better for a small number (≤ 4) of processors when compared to the other schemes, but does not scale well as more processors are used. This is because each processor needs to access all of array `B` multiple times (once per `i` iteration). The size of the shadow region needed is equal to the size of the array `B`. The communication code generated by the compiler copies `B` into the shadow region of each processor before the parallel loop. Although the resulting shadow region memory accesses are faster than remote memory accesses, the cost of broadcasting `B` becomes large when the number of processors increases. In addition, the number of times each processor is required to access `B` is inversely proportional to the total number of processors used (fewer `i` iterations). Consequently, as the number of processors increases, the reuse of the shadow data decreases and with it the performance advantage of using the shadow policy.

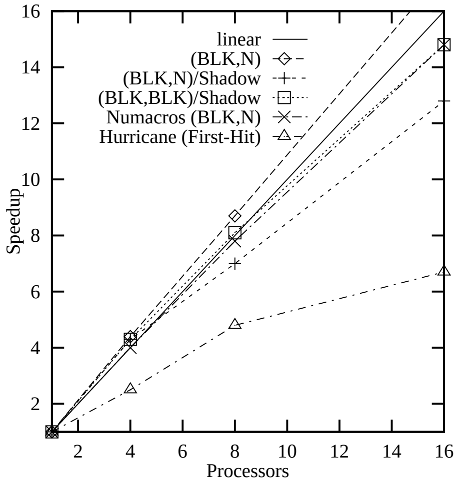
The performance of the LU kernel is shown in Figure 15(e). The speedup of the kernel using a block distribution is extremely poor because of the load imbalance caused by the triangular iteration space of the parallel loop in the kernel. The cyclic distribution performs better since it improves load balance. We have shown only the row-cyclic distribution since column-cyclic distributions cause false sharing, which degrades performance, as seen in Jacobi above. The performance using a shadow region with the row-cyclic distribution is better than that using the operating system cache coherence policy with the same distribution. Similar to Jacobi, this is because of false sharing.

However, the performance of the LU kernel using shadow regions does not scale well with increasing number of processors. This is mainly due to the size of the shadow region generated by the compiler, which is equal to the size of the portion of the array `a` being updated in the Gauss-Jordan elimination step. This large shadow region is copied once in each `k` iteration of the kernel; i.e, `n-1` times, which affects performance negatively.

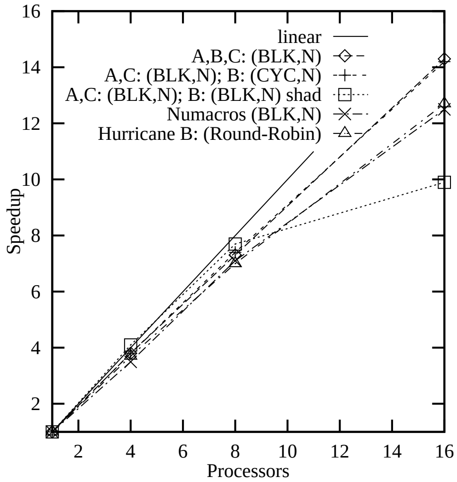
The performance of the Simplex kernel is shown in Figure 15(f). The use of shadow regions results in better performance than the operating system policies. This is attributed to the false sharing introduced by the operating system page-based cache coherence policy.

In Figure 16, the performance of the parallel programs generated by our compiler is compared to the performance of the parallel programs generated by `Numacros`, as well as to the performance of manually parallelized programs that use the default Hurricane memory management policies. The figure shows the results for the data size of 512×512, which is the only size for which `Numacros` results are available. For each kernel, the figure compares the performance obtained by our compiler to the best performance obtained by `Numacros` and to the best performance using the Hurricane policies. `Numacros` results are not available for the Simplex kernel. The figure shows that the performance of our compiler generated programs are significantly better than the performance of the Hurricane-based programs. The figure also shows that the best performance obtained by our compiler is better than that of `Numacros`. However, the `Numacros` performance is obtained manually matching loop and data partitioning; in the case of our compiler, loop partitioning is automatically determined by the compiler.

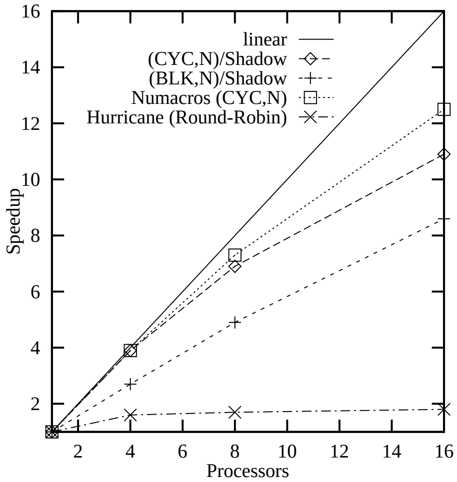
The impact of memory overhead reduction optimizations on the performance of the Jacobi kernel can be seen in Figure 17. The figure shows the speedup of the Jacobi kernel using a `(NIL,BLOCK)` decomposition scheme for both arrays `a` and `b`, which requires the use of an image array for global reference translation. The figure shows the speedup of the same kernel after each of the various optimizations described in section 3.7 is applied. Dimension swapping eliminates the need for the image array without introducing any overhead, and hence, results in extremely good performance. Dimension splitting also eliminates the image array, but introduces overhead in array index calculations. This overhead results in worse performance than that obtained using the image array. The use of dimension splitting requires the use of integer division. This may be implemented using a `div` operator, or when the divisor is a power of two, a less expensive logical-shift operator. The impact of the different costs can be clearly



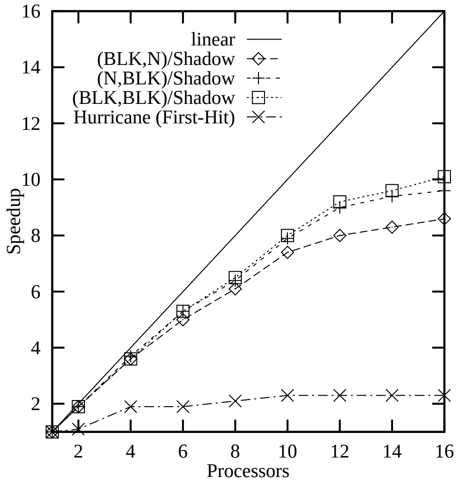
(a) Jacobi (512 × 512)



(b) Matrix Multiply (512 × 512)



(c) LU Factorization (512 × 512)



(d) Simplex (257 × 513)

Figure 16: Performance comparisons.

seen in performance. Finally, loop peeling can be used to break the first loop of the kernel into three loops such that remote region binding can be used in each loop, eliminating the need for the image array. This results in good performance, but the increase in loop overhead makes the optimization less effective than dimension swapping. Hence, the above results suggest that unless a very fast integer division operator is available, dimension swapping and loop peeling should be used to eliminate the overhead of using an image array.

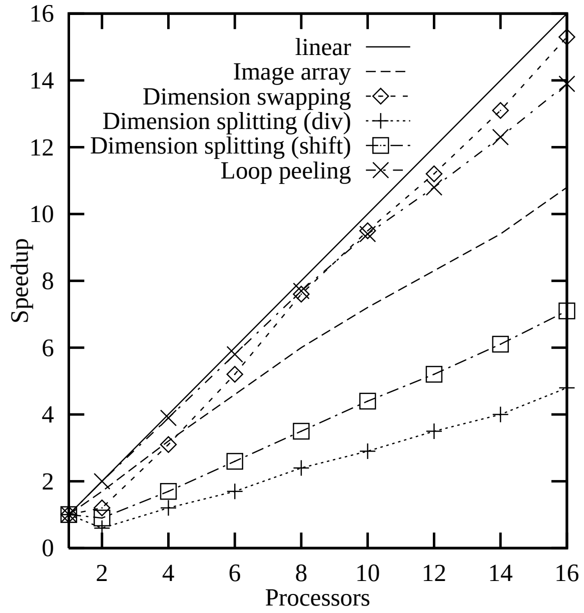


Figure 17: Impact of memory overhead reduction optimizations on performance.

5 Related work

A number of research and production compilers have been built to compile a data-parallel program into an SMPD parallel program for distributed memory multiprocessors. They include Fortran D [6], High-Performance Fortran [7], SuperB [19], Vienna Fortran [3], and Kali [12]. These compiler share the common approach of using programmer-supplied directives for specifying array decomposition schemes. Our compiler shares the same approach, but targets NUMA shared memory multiprocessors, hence, extending the applicability of the approach beyond distributed memory multiprocessors. The compiler is unique in the way it performs address translation taking advantage of the existing shared address space.

A number of approaches have been proposed to manage shared data on NUMA multiprocessors. These include operating systems page placement policies [8, 9], hierarchical clustering of operating systems [15], run-time libraries [13] and user-level macros [10]

Operating system techniques [8, 9, 15] fall short of exploiting the fine-grain parallelism that exists parallel applications. Operating systems are blind to application data access patterns and, hence, fail to enhance locality over a wide range of applications [9]. Even with knowledge of data access patterns, operating system policies deal with data in units of pages. The page is too coarse of a grain to accommodate the fine-grain sharing that occurs in parallel applications.

Sandhu, Gamsa and Zhou [13] address the issue of granularity and propose *shared regions* as a paradigm for data management on NUMA shared memory multiprocessors. Programmers annotate their parallel code by specifying the regions accessed by a thread as part of the synchronization primitives in the program. A run-time system migrates and replicates data across the physical memories in the system to enhance locality. Shared regions can lead to good performance, but require programmers to first parallelize and then annotate applications.

Li and Sevick [10] describe a set of C macros that implement data distribution and loop partitioning for improved performance on NUMA multiprocessors. Their approach is similar to ours in that they use data decompositions to achieve locality. However, they require programmers to manually insert loop partitioning directives in addition to data partitioning directives, and they use array allocation strategies that lead to additional index calculation. They propose compiler optimizations to reduce the impact of these calculations. In addition, they ignore cache performance by flushing cache contents after each parallel loop. In contrast, we provide shadow regions as a mechanism for improving cache performance; data partitions remain cacheable, only temporary buffer are flushed out of the cache.

6 Conclusions

Management of program data to reduce false sharing and improve memory locality is important for scaling performance of applications on NUMA shared memory multiprocessors. However, restructuring programs to manage data is tedious and error-prone. In this paper, we suggested and demonstrated that data management in array-based scientific applications can be effectively performed using compilers, making it easy for programmers to manage data.

We have developed a compiler system that accepts as input a data parallel program in which parallelism is expressed using the `FORALL` statement, as well as data decomposition schemes for arrays. The compiler generates an SPMD program in which parallelism is expressed using appropriate operating system calls, and in which arrays are partitioned and placed in the physical memory modules of the multiprocessor. The compiler takes advantage of the underlying shared memory architecture to efficiently implement array partitioning and distribution. In addition, the compiler provides a cache-coherence scheme based on shadow regions to avoid false sharing. Experimental results obtained from the Hector multiprocessor indicated that the compiler-generated parallel programs have significantly improved performance compared to parallel programs which rely on operating systems polices for data management.

The techniques presented in this paper are applicable to other multiprocessor platforms as well. The performance of applications on shared memory multiprocessors with hardware support for cache coherency can be improved by reducing false sharing that occurs at the cache-line level and by enhancing memory locality. The techniques are also applicable on networks of workstations that employ virtual-shared memory to provide programmers with a single coherent view of memory. On such systems, our techniques can be expected to significantly improve performance as interprocessor communications is performed using costly message passing.

The study indicates that the selection of a data decomposition scheme is critical for scaling performance. However, the process of arriving at the most suitable data distribution can be tedious in large programs. Hence, one extension to this work is to enhance the compiler with a static performance estimator that provides programmers with feedback on the quality of selected distributions. A second extension is to address the support for dynamic data decompositions to accommodate changing data access patterns during run-time. Another useful extension is to consider compiler algorithms for automatically deriving data decompositions.

References

[1] T. Abdelrahman and T. Wong, "Distributed array data management on NUMA multiprocessors," *Proceedings of Scalable High Performance Computing Conference,* pp. 551–559, 1994.

[2] V. Balasundaram, "A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor," *Journal of Parallel and Distributed Computing,* vol. 9, no. 6, pp. 154–170, 1990.

[3] B. Chapman, P. Mehrotra and H. Zima, "Vienna Fortran– a language extension for distributed memory multiprocessors," Report # 91–72, ICASE, 1991.

[4] Convex Computer Corporation, Convex Exemplar System Overview. Richardson, TX, USA, 1994.

[5] Cray Research, The T3D massively parallel processor system, Cray Research, 1993.

[6] S. Hiranandani, K. Kennedy and C. Tseng, "Compiling Fortran D," *Comm. of the ACM,* vol. 35, no. 8, pp. 66–80, 1992.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification, Report # CRPC-TR92225, Center for Research on Parallel Computation, Rice University, 1992.

[8] R. LaRowe, C. Ellis, and L. Kaplan, "Tuning NUMA memory management for applications and architectures," *Proceedings of the Symposium on Operating System Principles,* 1991.

[9] R. LaRowe, J. Wilkes and C. Ellis, "Exploiting operating system support for dynamic page placement on NUMA shared memory multiprocessors," *Proc. of the Third ACM PPOPP Symp.,* pp. 122–132, 1991.

[10] H. Li and K. Sevcik, "NUMACROS: Data Parallel Programming on NUMA Multiprocessors," *Proc. of 4th Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 247–263, September 1993.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz and M. Lam, "The Stanford Dash multiprocessor," *IEEE Computer,* vol. 25, no. 3, pp. 63–79, 1992.

[12] P. Mehrotra and J. Van Rosendale, "Compiling high level constructs to distributed memory architectures," *Proc. of the 4th Conf. on Hypercube Conc. Computers and Applications,* 1989.

[13] H. Sandhu, B. Gamsa and S. Zhou, "The shared region approach to software cache coherence on multiprocessors," *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pp. 229–238, 1993.

[14] D. Solow, *Linear programming: An introduction to finite improvement algorithms,* North-Holland, New York, 1984.

[15] M. Stumm, R. Unrau and O. Krieger, "Clustering micro-kernels for scalability," *USENIX Workshop on Micro-Kernels,* pp. 285–303, 1992.

[16] Z. Vranesic, M. Stumm, R. White and D. Lewis, "The Hector multiprocessor," *IEEE Computer,* vol. 24, no. 1, 1991.

[17] T. Wong, Data partitioning based compiling techniques for NUMA shared memory multiprocessors. M.A.Sc. thesis, Department of Electrical and Computer Engineering, University of Toronto, 1994.

[18] H. Zima and B. Chapman, *Supercompilers for parallel and vector computers,* ACM Press, New York, 1990.

[19] H. Zima, H. Bast and M. Gerndt, "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing,* vol. 6, no. 1, pp. 1–18, 1988.