

# Distributed Array Data Management on NUMA Multiprocessors\*

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

## Abstract

*Management of program data to reduce false sharing and improve locality is critical for scaling performance on NUMA multiprocessors. We use HPF-like directives to partition and place arrays in data-parallel applications on Hector, a shared-memory NUMA multiprocessor. We present experimental results that demonstrate the magnitude of the performance improvement attainable when our proposed array management schemes are used instead of the operating system management policies. We then 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.*

## 1 Introduction

In order to achieve scalability, state-of-the-art shared memory multiprocessors are employing a hierarchical memory structure in which shared memory is physically distributed, and a scalable low latency interconnection network is used to support distributed-shared memory. This structure makes memory access time dependent on the distance between processor and memory. Hence, these multiprocessors are referred to as Non-Uniform Memory Access (NUMA) multiprocessors. Examples of NUMA multiprocessors include the Stanford Dash [7], the KSR1 [5], and Toronto's Hector multiprocessor [12].

NUMA multiprocessors are programmed using a shared memory programming model. In this model, application programmers are not concerned with the

management of data in shared memory. Tasks such as the placement, partitioning, replication and migration of data across the physical memories in the system are typically delegated to the operating system, as a part of its page management policies [6, 11].

In this paper, we advocate that user-level data management is as important in NUMA multiprocessors as in distributed memory multiprocessors. Operating system policies deal with application data at the page level. This is too coarse of a grain to exploit locality of reference that exists in applications due to fine grain sharing of data, and leads to considerable false sharing. Data must be partitioned, placed and possibly replicated in order to enhance data locality and minimize false sharing. We present supporting experimental results from the Hector multiprocessor using four application kernels: *matrix multiplication*, *LU factorization*, *Successive Over Relaxation (SOR)*, and *Simplex* linear optimization. The results indicate that in all four applications, the partitioning, placement and replication of data across the physical memories of the multiprocessor have a strong impact on performance. The results also suggest that the most suitable scheme for partitioning data depends not only on the nature of the application, but also on the number of processors and the size of the data; parameters that sometimes are not known until run-time.

Delegating data management to the programmer has its drawbacks, nonetheless. Typically, it is not apparent to the programmer how to partition program data, nor how to replicate shared data in order to minimize the execution time of an application. The programmer is forced to experiment with several data partitioning, data placement and data replication schemes to arrive at the most suitable one. In each case, the programmer must restructure the application program in order to reflect the particular scheme used. This is a time consuming and an error prone process that reduces programmer's productivity.

---

\*This work has been supported by NSERC and ITRC research grants.

In order to automate the tasks of data management we have designed a compiler system for the Hector multiprocessor. The compiler targets large-scale scientific applications that use large regular arrays of data in nested loops. The compiler accepts directives for regular partitioning and distribution of arrays, similar to those used in High Performance Fortran [3]. The compiler’s analysis of input programs is largely based on compilation techniques for distributed memory multiprocessors. However, the compiler takes advantage of the shared memory architecture of Hector to efficiently implement and manage storage for distributed arrays.

The remainder of this paper is organized as follows. In section 2 an overview of the Hector multiprocessor is given. In section 3 the four applications we use in our study and the results of their parallelization on Hector are described. In section 4 our compiler system and results from a prototype implementation are described. In section 5 relevant work is reviewed. Finally in section 6 concluding remarks are presented.

## 2 The Hector multiprocessor

Hector [12] 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 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. Hector has no hardware mechanisms for cache consistency. Our experiments have been conducted on a 16-processor cluster consisting of 4 stations on a single local ring.

Hurricane [11] 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 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 shared 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 proto-

col for each 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 as 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 operating system also controls the placement of data in the physically distributed shared memory via two page placement policies. The *First-hit* policy places a page in the 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 The experiments

In this section the four application kernels used in our experiments are described, and the results obtained from the Hector multiprocessor are presented. The application kernels are: matrix multiply, LU factorization, SOR and Simplex.

**Matrix Multiply.** This is a standard matrix multiplication kernel that computes the product of two square matrices **a** and **b** as follows:

```
for i=1,n
  for k=1,n
    r = a[i][k] % register allocated
    for j=1,n
      c[i][j] += r*b[k][j]
```

The kernel is parallelized by having each processor compute an equal portion of the output matrix **c**.

**LU factorization.** This kernel decomposes an  $n \times n$  square matrix **a** into a lower and an upper triangular matrices using Gauss-Jordan elimination with pivoting and row exchange. The main part of the kernel can be expressed as follows:

```
for k=1,n-1
  % find index of pivot
  ipivot =index(max(a[k][k],n-k))
```

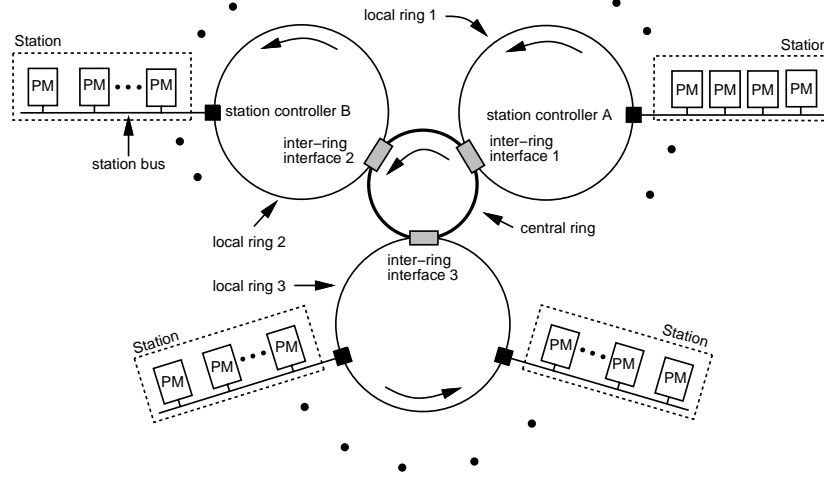


Figure 1: The Hector multiprocessor.

```

% exchange pivot and current rows
exchange(k,ipivot)
for 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]

```

The kernel is parallelized by having each processor perform the Gauss-Jordan elimination computations on an equal portion of the submatrix below and to the right of the pivot in each of the  $k$  iterations.

**SOR.** The successive-over-relaxation kernel is an algorithm for iteratively solving partial differential equations. The main part of the kernel is shown below.

```

repeat
    for i=2,n-1
        for 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
    for i=2,n-1
        for j=2,n-1
            a[i][j]=b[i][j]
until convergence

```

This kernel is parallelized by assigning an equal subset of the array  $b$  to each processor.

**Simplex.** This application kernel solves an  $m \times n$  linear programming problem using the well-known simplex

method. The method uses an  $(m + 1) \times (n + 1)$  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 [10]. The main part of each iteration is:

```

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

```

The kernel is parallelized by having each processor perform an equal subset of the Gauss-Jordan elimination computations. One processor is designated as a master, and in addition, it performs the computations necessary to find the pivot column and the pivot row in each iteration of the kernel.

The performance of the four applications on Hector is shown in Figure 2. The performance is reported in terms of the speedup of each application, which is defined as the ratio of the sequential execution time to the parallel execution time of the application.

The curve labeled “Hurricane” shows the speedup of each application using Hurricane’s page-based cache consistency protocol and data placement policies. The

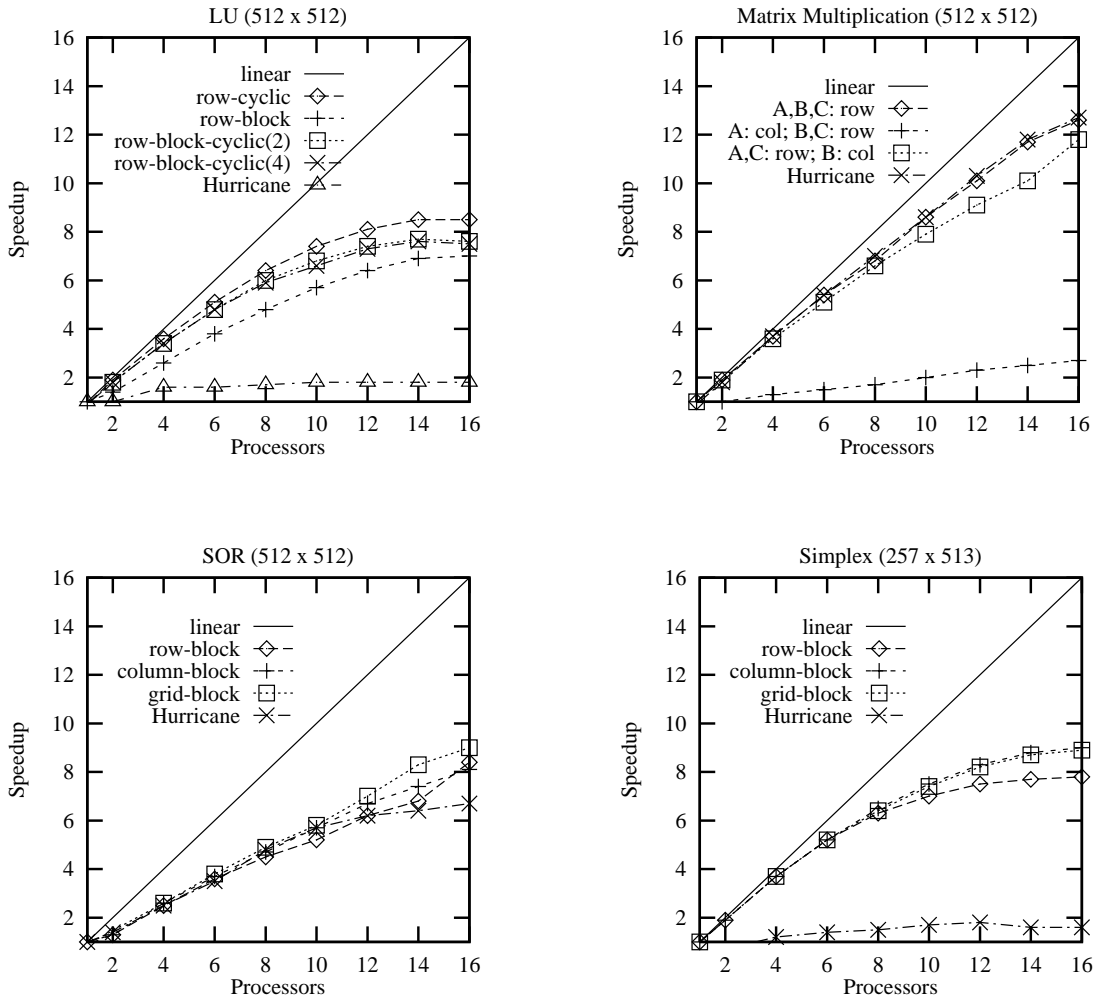


Figure 2: The performance of the four application kernels.

poor performance indicated by these curves is attributed to false sharing and poor data locality. The relatively large size of a page (4 Kbytes) makes many data objects not shared by processors reside on the same page. Independent references to the data by processors cause the page to become uncacheable, which increases access time to data and degrades performance. In the case of matrix multiply, the  $\mathbf{a}$  and  $\mathbf{b}$  matrices are both read-only, and hence, false sharing is not an issue. The operating system page placement policies do not take into consideration the data locality needed to achieve scaling performance. In all the applications, one of the processors is initially responsible for program I/O and initializations. Consequently, the majority of program data is either placed in one physical memory under the first-hit policy, or placed scattered across the physical memories under

the round-robin policy. In both cases, data is not necessarily located in the physical memory of the processor using the data the most. This results in a large number of non-local memory references and hence, a degraded performance.

The remainder of the curves for each application show the performance of the application after the program has been *manually* restructured to eliminate false sharing and enhance data locality. The programs have been restructured to partition the data according to how the data is used by the processors, and place the pages containing data in the memory of the processor that uses the data the most. The various data partitioning schemes will be described in the next section. Shared data (such as the pivot row in the Simplex kernel, for example) has been replicated to further avoid false sharing. The curves reflect significantly better

performance, which indicates the importance of eliminating false sharing and enhancing data locality.

The results also indicate that the data partitioning scheme has an impact on performance. For example, column-block partitioning outperforms row-block partitioning for the Simplex application. This is attributed to the number of non-local memory references incurred in each scheme, and to the impact the scheme has on the communication patterns among the processors. In the Simplex application, a processor makes  $O(\mathbf{n})$  non-local memory references to obtain the pivot row under row-block partitioning. However, only the master processor makes  $O(\mathbf{m})$  non-local memory references under column-block partitioning. This results in a smaller number of non-local memory references since  $\mathbf{m} \ll \mathbf{n}$  in our linear programming problems. In addition, since all the processor share the pivot row, all processors must access a single physical memory to obtain a copy of that row under row-block partitioning. This creates a “hot-spot” at which the processors contend to access memory. However, under column-block partitioning, the pivot row becomes distributed among the processors, levitating the hot-spot.

Finally, the results indicate that the best scheme for data partitioning depends on the number of processors. For example, in the Simplex kernel, row-block partitioning results in slightly better performance than column partitioning up to 4 processors, but in worse performance for a larger number of processors. There is also similar dependence on the size of the data, and the placement of data partitions on the processors. However, due to space limitations, these results are not presented here [13].

## 4 The compiler

### 4.1 Overview

The results presented in the previous section indicate that user-level data management is essential for achieving scaling performance on NUMA multiprocessors. However, the task of re-structuring programs to partition, place and replicate data is a tedious and error-prone one. Hence, we have designed and implemented a prototype of a compiler system on Hector which automates the task of restructuring programs to reflect desired array partitioning and distribution schemes.

The compiler takes as input programs written in a restricted version of the C programming language [13]. Parallelism in an input program is expressed using

**forall** statements. The compiler also takes as input a scheme of data partitioning for each array in a parallel loop nest and a specification of processor geometry. The compiler produces a Single Program Multiple Data (SPMD) parallel program for Hector. In this parallel program, arrays are partitioned, and the partitions are placed in the physical memories of the processors according to the specified partitioning schemes. Parallelism in the output program is expressed using the appropriate operating system calls that create, manage and synchronize light-weight processes. Each process executes the subset of the iterations of parallel loops in the input program.

An array is partitioned by assigning a partitioning attribute to each dimension of the array. The partitioning attributes supported by the compiler are those supported by the HPF language [3]. The **BLOCK** attribute partitions the array in a dimension in equal size blocks such that each processor has a contiguous block of the array in that dimension. The **CYCLIC** attribute partitions the array by assigning elements of the array in a dimension to the processors in a round-robin fashion. The **BLOCK-CYCLIC** attribute first groups array elements in a given dimension in contiguous blocks of equal sizes, and then **CYCLIC**-ly partitions the blocks. The **\*** attribute indicates that the array is not partitioned in a dimension. Hence, for a two-dimensional array, **(BLOCK,\*)** specifies row-block partitioning, **(\* ,CYCLIC)** specifies column-cyclic partitioning, and **(BLOCK,BLOCK)** specifies grid-block partitioning of the array.

The processor geometry specifies an arrangement of the processors in the form of a multi-dimensional mesh. This allows the compiler to determine the number of processors allocated to each dimension of a partitioned array, and hence, to map array partitions onto processors.

The compiler consists of two main phases: program analysis and parallel code generation. The two phases are described in the sections below. The description focuses on the unique aspects of our compiler and its implementation of distributed arrays.

### 4.2 Program analysis

The purpose of program analysis is to map array partitions to processors, to determine the subset of parallel-loop iterations executed by each processor, and to classify array references that appear in parallel loops as either local or non-local. This is done using the array partitioning schemes and the processor geometry specified in the input program, and by applying the owner-compute rule [4]. The analysis is

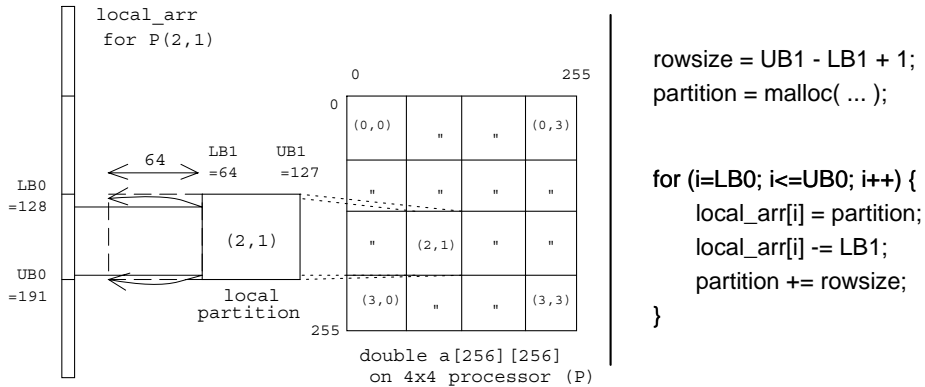


Figure 3: Local partition allocation.

largely based on that of Fortran D [4], and hence, will be only overviewed here. The analysis consists of three main steps. In the first step, the array partitioning schemes and the processor geometry are used to determine a local array index set for each partitioned array on each processor. This defines the portion of each array that is to become local in the physical memory of each processor. In the second step, the local array index sets and the subscript expressions in LHS array references in a parallel loop are used to determine a loop index set for that loop on each processor. This set describes the subset of the iterations of that loop to be executed by each processor. In the third step, loop index sets and the subscript expressions in RHS array references in a parallel loop are used to classify the RHS array references. References that only access array elements in the local partition of the array are referred to as *local references*; references that access array elements in non-local partitions of the array are referred to as *remote references*. Unlike program analysis for distributed memory machines, there is no need to translate array index sets into local index sets for each processor since arrays reside in globally shared memory.

### 4.3 Parallel code generation

The parallel code generation phase of the compiler generates code to allocate in each physical memory a local region for each array partition that is to reside in that memory. It also generates code to create, coordinate and synchronize light-weight processes which execute iterations of `forall` loops in parallel as determined by program analysis.

The distributed allocation of arrays gives rise to two problems. First, a reference to an array element must be made into an equivalent reference within the region containing the array element in either local or

remote memory. This is necessary since the array is segmented in shared memory, yet is accessed by the program as if it were stored in a contiguous form. We refer to this as *reference translation*. Second, the cacheability of local array partitions must be maintained, even when some of the array elements in a local partition are accessed by more than one processor. This is necessary to assure that in during the parallel execution of a loop nest, data accessed by a processor is in a cacheable state. The compiler uses operating system calls to restore the cacheability of pages that become uncacheable.

The need to translate array references and to maintain cacheability can result in overhead that affects the performance of a parallel program, and hence, must be performed efficiently. The compiler takes advantage of the shared memory of Hector and uses pointer manipulation and a number of memory allocation schemes to efficiently translate references and maintain the cacheability of distributed arrays. The suitability and performance of each scheme depends on the partitioning scheme of the array.

In order to make the translation of local references efficient, the pointer to the local region of the array is offset by a constant amount that depends on the size of the local region and the relative position of the region in the array. This is shown in Figure 3 for a  $256 \times 256$  array with **BLOCK** partitioning in both dimensions and a  $4 \times 4$  processor geometry. Local references to the local portion of an array are performed with no overhead.

The translation for remote references can be done in a number of ways. If the data partitioning scheme is such that all but one of the dimensions of an array reside in one physical memory, then remote regions of the array can be accessed by directly using the pointers to these regions. We refer to this scheme as *remote*

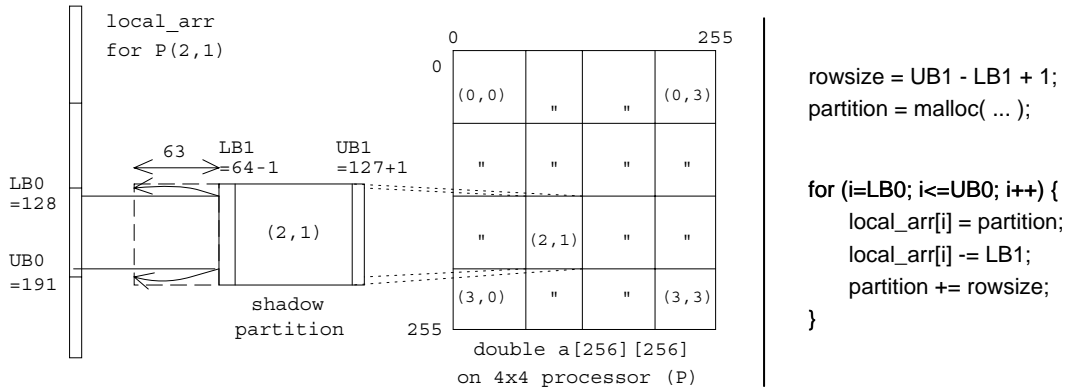


Figure 4: Local partition allocation with shadow regions.

*region binding*. This is the simplest scheme since it requires no additional pointer processing or memory allocation. However, the scheme makes it prohibitive to maintain the cacheability of array partitions during the execution of a parallel loop. Since remote array elements are accessed directly, the cacheability of the pages containing these elements has to be restored on every access. Since this is done using expensive operating system calls, the cost of maintaining cacheability becomes prohibitive.

If remote region binding is not possible, then an *image* array is used to translate remote references. The image array is a physically contiguous array of pointers of the same size as the distributed array, and which is initialized so that reference to an element in the image array give the address of corresponding element of the distributed array. All references to the distributed array in the body of a parallel loop are replaced by a de-reference operation of the corresponding element in the image array. This translation scheme adds overhead to each remote reference due to the additional memory accesses to the image array. This overhead is incurred regardless of whether the element accessed by the reference is in the local or a remote array partition. However, since the image array is only read by the processor after it has been initialized by the compiler, it can be made cacheable to all the processors. If a large portion of the image array remains in the cache, the resulting overhead becomes minimal. The use of the image array, nonetheless, adds considerable memory space overhead to the program. The scheme has the same problems in maintaining cacheability as remote region binding.

Reference translation can also be performed by using a *shadow region*. The local region allocated in each memory is enlarged to hold both local array elements and a copy of the remote array elements referenced

by the processor. The part of the region used to hold the copy of the non-local elements is the shadow region. This is shown in Figure 4 for a  $256 \times 256$  array with **BLOCK** partitioning in both dimensions and a  $4 \times 4$  processor geometry. Code is inserted at the beginning of a parallel loop to copy the data from remote memories into local shadow regions before the execution of the parallel loop nest. Remote array references become references to the local shadow region, and are performed with no overhead. This scheme adds the overhead of data copying and uses additional memory space. Since array partitions must be rectangular, there can be significant memory waste, especially if the remote elements are not in the immediate spatial neighborhood of the data local to a processor. However, it makes it easy to maintain cacheability. Operating system calls are inserted after the shadow regions have been copied to restore the cacheability of both the array partitions and the shadow regions. This code is executed only once per parallel loop for all remote references within the loop.

Remote reference translation can be also performed by using loop transformations such as loop peeling and/or loop splitting [14]. A parallel loop nest is split into multiple nests, such that in each nest array references are either local or are directed to only one remote memory, similar to remote region binding. This allows the replacing array references in each nest by ones that directly reference the appropriate local or remote array partition. This translation method, however, is not possible in all applications, and when possible, can increase program overhead due to the increased number of loop nests in the program.

#### 4.4 Performance

The performance of the compiler-generated parallel kernels on Hector is shown in Figure 5 using a number

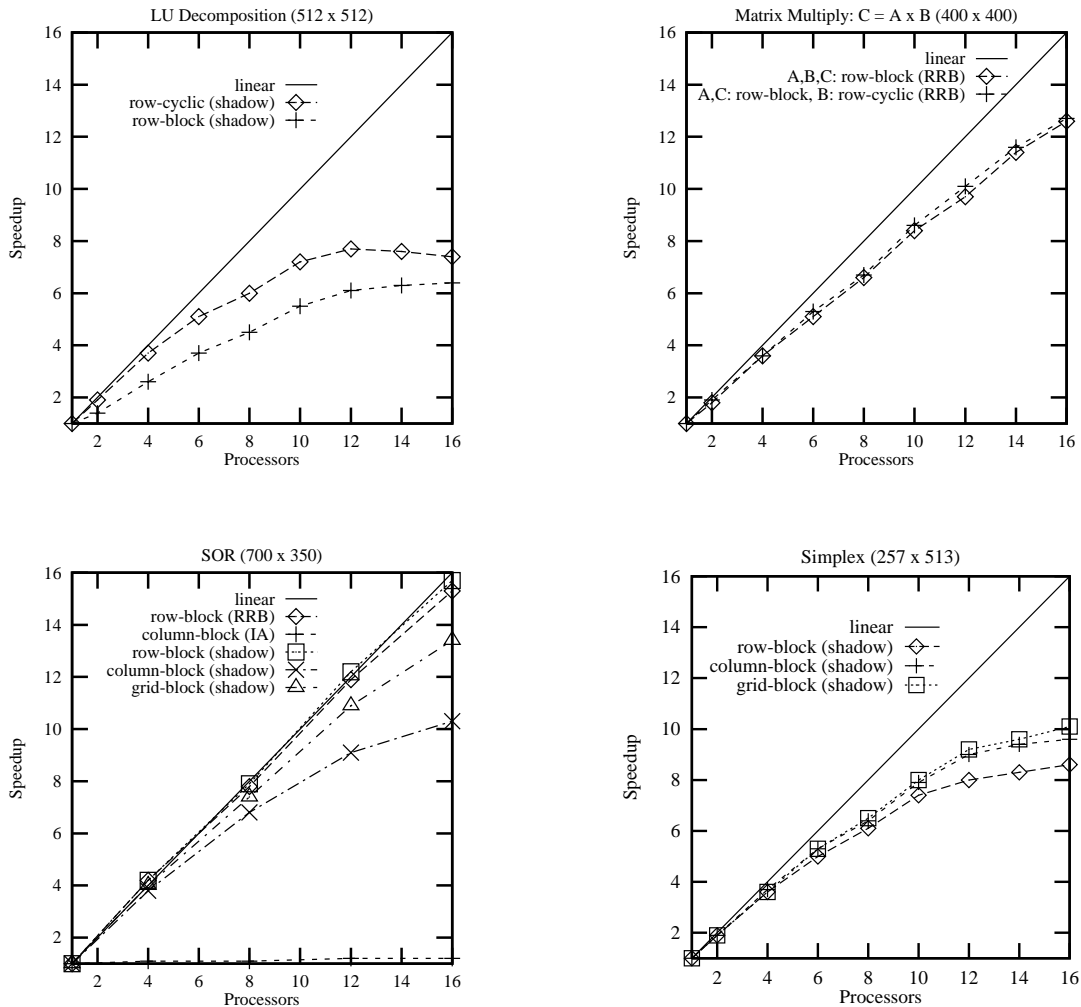


Figure 5: Performance of the compiler-generated parallel kernels.

of array partitioning and memory allocation schemes. **RRB** refers to remote region binding, **IA** refers to image array, and **shadow** refers to shadow regions. The results indicate that relatively scaling performance can be obtained using the compiler on the applications.

In the matrix multiplication kernel, remote region binding is used for both the **A** and **B** arrays. Since both arrays are only read by the processors, cacheability need not be maintained. The performance of the kernel is only limited by its memory access patterns.

The performance of the SOR kernel can be used to indicate the relative merits of the various translation schemes described in the previous section. The best performance is obtained by using the **BLOCK** attribute on the rows of the arrays, and using shadow regions to maintain cacheability. Similar performance can be obtained by using remote region binding. The lack

of cacheability in this case does not significantly affect performance due to the small number of pages that become uncacheable when the partitioning is row-block. This is in contrast to the performance of the kernel when column-block partitioning an an image array are used. The number of pages that become uncacheable is large, and that adversely affects performance.

The performance of the LU kernel does not scale well. This is attributed to the row exchange step in each iteration of the outer loop in the kernel. The two rows that are exchanged may reside in different processors. Hence, it becomes necessary to restore the cacheability of the affected pages each iteration. The high cost of the operating system calls that restore page cacheability in the sequential section of the kernel degrades its performance. The performance of the Simplex kernel is affected by similar factors.

## 5 Related work

A number of research groups have proposed implementing high-performance compilers for distributed memory multiprocessors based on user-specified array partitioning and distribution directives [1, 2, 4, 8, 9]. Our work extends the applicability of these directives to NUMA multiprocessors with shared address spaces. The work has specifically addressed efficient implementations of array partitioning that take advantage of the available shared address space. This has not been addressed in previous work.

## 6 Conclusions

In this paper we have presented experimental results which indicate the importance of user-level data management on NUMA multiprocessors. User-level data management allows users to control how data is partitioned, placed and replicated in a physically distributed shared memory. This significantly improves performance by eliminating false sharing and increasing data locality.

We have developed a compiler system to automate the mundane tasks of data management. The compiler accepts as input a data parallel program in which parallelism is expressed using the `forall` statement, as well as partitioning and placement schemes for arrays. The compiler then generates a SPMD program in which parallelism is expressed using appropriate operating system calls, and in which data partitioning and placement schemes are implemented. The compiler takes advantage of the underlying shared memory architecture to efficiently implement array partitioning and distribution. Future work will address the development of performance models to provide the compiler with the ability to estimate the goodness of data distributions. Future work will also address the support for dynamic data distributions to accommodate changing data access patterns during run-time.

## References

- [1] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, "A static performance estimator to guide data partitioning decisions," *Proc. of the 3rd ACM PPOPP Symp.*, pp. 213–223, 1991.
- [2] B. Chapman, P. Mehrotra and H. Zima, "Videnna Fortran— a language extension for distributed memory multiprocessors," Report # 91-72, ICASE, 1991.
- [3] High Performance Fortran Language Specification, Report # CRPC-TR92225, Center for Research on Parallel Computation, Rice University, 1992.
- [4] S. Hiranandani, K. Kennedy and C. Tseng, "Compiling Fortran D," *Comm. of the ACM*, vol. 35, no. 8, pp. 66–80, 1992.
- [5] KSR1 Technical Summary, Kendall Square Research Corporation, Boston, MA, 1992.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] A. Rogers and K. Pingali, "Process decomposition through locality of reference," *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 69–80, 1989.
- [10] D. Solow, *Linear programming: An introduction to finite improvement algorithms*, North-Holland, New York, 1984.
- [11] M. Stumm, R. Unrau and O. Krieger, "Clustering micro-kernels for scalability," *USENIX Workshop on Micro-Kernels*, pp. 285–303, 1992.
- [12] Z. Vranesic, M. Stumm, R. White and D. Lewis, "The Hector multiprocessor," *IEEE Computer*, vol. 24, no. 1, 1991.
- [13] T. Wong, A tool for data management on Hector, M.A.Sc. thesis, University of Toronto, *in preparation*, 1994.
- [14] H. Zima and B. Chapman, *Supercompilers for parallel and vector computers*, ACM Press, New York, 1990.