Evaluation of Dynamic Data Distributions on NUMA Shared Memory Multiprocessors

Tarek S. Abdelrahman and Kenneth L. Ma
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada M5S 3G4

Abstract

Dynamic data distributions offer a number of performance benefits, but require more sophisticated compiler support and incur run-time overhead. We investigate attainable benefits using a compiler system we developed for the Hector NUMA multiprocessor. We show that the benefits depend on a number of factors, including data size relative to the cache size, data access patterns, the degree of "NUMAness" of the multiprocessor system, and the extent to which data is reused. Programmers and compiler designers must take these factors into consideration.

1 Introduction

Non-Uniform Memory Access (NUMA) multiprocessors have become widely available in the last few years; examples include the KSR1/2 [8], the CRAY T3D [6], the Convex Exemplar [5], and Toronto's Hector [9]. NUMA multiprocessors can scale to large numbers of processors while supporting shared memory programming. Nonetheless, the non-uniform nature of memory accesses requires careful management of data in shared memory to enhance data locality. Recent work [1, 2] has shown that data distributions [7] provide a good abstraction for compiler-based locality management on NUMA multiprocessors. Programmers specify data distribution schemes to partition arrays in the program; the compiler uses this information to place array partitions across the memory modules, enhancing locality and eliminating false sharing.

Data distributions can be *static*, defined at compile time and remaining fixed during program execution, or *dynamic*, defined and possibly changed during program execution. Dynamic distributions are desirable in: (1) applications that have multiple phases of computation, to allow the most suitable distribution to be used in each phase of computation; (2) applications in which the best data distribution depends on run-time specified parameters, such as array sizes and number of processors; (3) applications that use sparse data, making the most appropriate distribution that balances workloads dependent on the nature of sparsity; and (4) in run-time parallel libraries called with several distributions.

Supporting dynamic data distributions requires more sophisticated compile-time analysis [4]. There can also exist considerable run-time overhead of the data transfers needed to re-locate data in memory modules, and of run-time partitioning of parallel loops. It is unclear to what extent this overhead degrades performance, and whether the performance benefits outweigh the overhead. Although compilers systems such as HPF [7] and Vienna Fortran [4] support dynamic data distributions in their language specification, there has been little experimental evaluation of their benefits and overhead.

In this paper, we describe our experiences with dynamic data distributions on the Hector shared memory multiprocessor [9]. In the past, we have implemented a compiler system to support user-specified data distributions on Hector [1]. In this work, we extend this compiler to support dynamic data distributions and use it to assess the benefits of dynamic data distributions for a number of synthetic benchmark programs and applications. We conclude that the benefits of dynamic data distributions generally outweigh their cost.

The remainder of this paper is organized as follows. Section 2 gives an overview of the analysis and run-time support needed for dynamic data distributions. Section 3 describes the experimental results obtained using the compiler on the Hector multiprocessor. Finally, Section 4 gives conclusions.

2 The compiler

The compiler system supports regular data distribution directives similar to the `DISTRIBUTE` and `REDISTRIBUTE` directives of HPF [7], and parallel `FORALL` loops. It translates an input program containing such directives into an SPMD program containing operating system calls for the creation and coordination of light-weight processes. This program can be compiled by the native compiler on the target machine. The output program also contains appropriate calls to a data distribution library (DDL) which performs run-time partitioning of arrays and loops.

Array distribution is implemented using a number of array allocation schemes [1, 2]. The compiler generates code to allocate a region in the local memory of each processor that holds each array partition that resides in this local memory. Such 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. Hence, array references in the data parallel program must be 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 with little run-time overhead.

An array *partition* descriptor is maintained at run-time for each distributed array on each processor. It contains the array size, the data structures needed to perform reference translation, and a description of the currently active array distribution scheme. The partition descriptor is initialized by the compiler and then updated by compiler-inserted code at run-time when the array is redistributed. Loop partitioning is performed using the owner-computes rule [7] at run-time since the actual distribution of the array may not be know until then. Hence, the compiler inserts calls to DDL before the outermost loop in a loop nest to determine (using the owner-computes rule) loop bounds on each processor given the active distributions of arrays referenced in the left hand side of loop statements. Since the compiler targets shared memory, it does not generate messages; it only inserts ownership tests and synchronization code when necessary.

Array data redistribution is performed in two steps in parallel by all processors. First, each processor allocates space for the array data in its local physical memory and initializes its run-time structures to reflect the new distribution scheme of the array. Second, each processor copies array data into the allocated space. The source of the data is determined using the old distribution scheme of the array. Barriers are used to synchronize the steps of redistribution.

In order to increase the efficiency of dynamic distributions, the complier determines the *reaching data distributions* [4] to each parallel loop nest. For example, if a loop nest is reached by identical data distribution schemes, then loop partitioning may be performed statically at compile time, avoiding run-time overhead. Determining reaching distributions is similar to determining reaching definitions [3], and is solved using data-flow analysis.

3 Experimental results

Hector [9] is a scalable shared memory multiprocessor which consists of a set of processing modules connected by a hierarchical ring interconnect. 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 portion of shared memory. Our experiments have been conducted on a 16-processor cluster consisting of 4 stations on a single local ring.

The first benchmark consists of two passes with orthogonal parallelism; the first pass has the `i` loop parallel and the second pass has the `j` loop parallel. Each pass reads elements of the array `a`, as shown in Figure 1(a). The outermost `k` loop in each pass results in data reuse; the larger the loop bound `Iter`, the more the array data is reused. A static row-block data distribution of the array `a` results in all local accesses to `a` in the first pass, but results in non-local accesses to the array in the second pass. Since the array is only read and not written, this is the only penalty of using a static distribution. Data redistribution may be used between the two passes to make all accesses to `a` in the second pass local. The performance benefit realized by data redistribution is depicted in Figure 2(a). The benefit is measured in terms of the ratio of the execution time of the application with a static data distribution to the execution time of the application with data redistribution. Hence, a ratio less

```
for k=1,Iter
 forall i=1,n
  for j=1,n
   b=b+a(i,j)/2;

for k=1,Iter
 forall j=1,n
  for i=1,n
   b=b+a(i,j)/2;
```

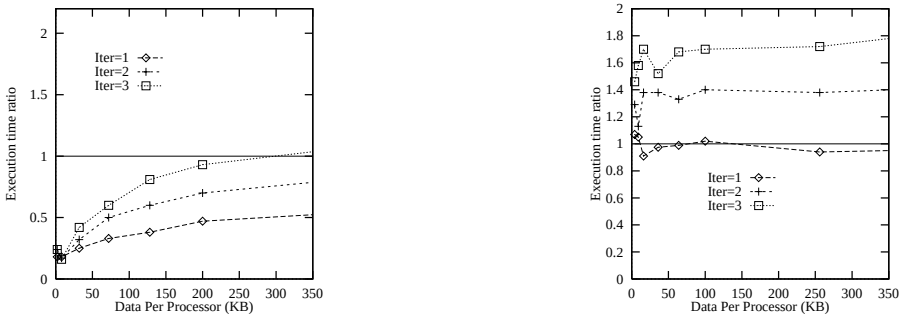
(a) Application I

```
for k=1,Iter
 forall i=1,n
  for j=1,n
   a(i,j)=(a(i,j-1)+a(i,j+1))/2

for k=1,Iter
 forall j=1,n
  for i=1,n
   a(i,j)=(a(i-1,j)+a(i+1,j))/2
```

(b) Application II

Figure 1: The benchmark applications.



(a) Application I (b) Application II

Figure 2: Performance benefits for the benchmark applications.

than 1 indicates performance degradation with data redistribution and a ratio greater then 1 indicates performance gains. The benefit is shown for various data sizes per processor and for various amounts of data reuse (different values of `Iter`). Figure 2(a) indicates that no benefit is attained from the use of data redistribution. This is because the cost of accessing remote memory in the second pass is small compared to the overhead incurred by re-distributing the data. However, the extent performance is degraded depends on the data size and the amount of reuse. When the size of the data is small enough to fit in the cache, a single remote memory access is sufficient to make it local for further accesses by the processor, making the redistribution completely unnecessary. However, when the size of the data is larger than the size of the cache, subsequent accesses to the data may be remote, and hence, redistributing the data yields more benefit, but not enough to outweigh the cost of redistribution.

The second benchmark is similar to the first one, but it reads and writes `a`, as shown Figure 1(b). If a static row-block distribution is used, the second pass of the benchmark must be executed in a "wavefront" fashion because of the cross-processor dependence carried by the `i` loop. Blocks of iterations of the `j` loop in the second pass are executed in each processor, and synchronization is used to stage the execution of the blocks across the processors. Hence, with a static distribution, the benchmark incurs not only remote memory accesses, but also the cost of synchronization. Data redistribution makes accesses in the second pass local and avoids synchronization. There is a significant benefit to performance when data redistribution is used, as shown in Figure 2(b). The benefit increases when the size of the data becomes comparable to the size of the cache size, and also increases when there is reuse of data.

The Hector prototype is configurable to add delays to off-station memory requests. This increases remote memory access latency, emulating a larger system or a system with a slower network. The impact of this delay on the performance of Application I is shown in Figure 3, when the delay increases off-station access time by 25%. It indicates that more benefit can be obtained from data redistribution when network latency is high.

Finally, the impact of using dynamic data distributions on the performance of real application, a 512×512 2-d FFT, is shown in Figure 4. The use of a single distribution for both phases of the application results in remote memory accesses and synchronization in the second phase of the application. The performance significantly benefits from data redistribution.

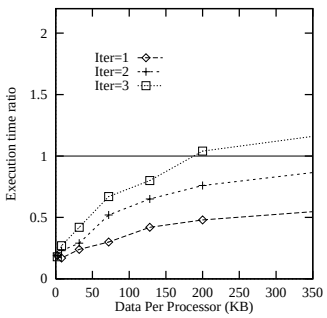


Figure 3: Impact of network delay

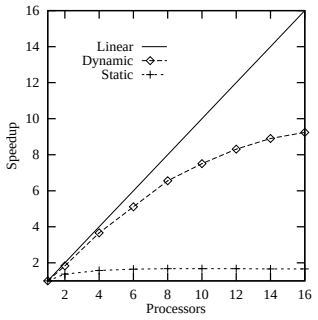


Figure 4: Performance benefits for 2-D FFT.

4 Conclusions

In this paper we examined the potential benefits of using dynamic data distributions on NUMA shared memory multiprocessors. We have implemented a compiler prototype that supports dynamic data distribution and experimented using a number of synthetic and real applications.

We conclude that data redistribution generally benefits application performance on NUMA multiprocessors when the working set of the application on a given processor does not fit in the local cache of the processor, either because of limited capacity or because of conflicts. Data redistribution also benefits performance when data access patterns in a phase of the computation is such that the use of the owner-computes rule results in overhead. Dynamic data distributions should not be used when the only penalty of using a static distribution is non-local memory accesses, nor when the working set of a processor fits and remains in its local cache because data locality is then maintained by the caches, making data redistribution unnecessary.

References

[1] T. Abdelrahman and T. Wong, "Distributed array data management on NUMA multiprocessors," *Proc. Scalable High-Performance Computing Conf.,* pp. 550-559, 1994.

[2] T. Abdelrahman and T. Wong, "Compiler support for distributed arrays on NUMA shared memory multiprocessors," Technical report CSRI-318, Computer Systems Research Institute, Toronto, Canada, 1995.

[3] A. Aho, R. Sethi and J. Ullman, *Compilers: principles, techniques and tools.* Addison-Wesley, Reading, Mass., 1985.

[4] B. Chapman, P. Mehrotra, H. Moritsch and H. Zima, "Dynamic data distributions in Vienna Fortran," Proc. Supercomputing, pp. 284–293, 1993.

[5] Convex Corporation, The Exemplar architecture, Convex Computer Corporation, 1994.

[6] Cray Research, The Cray research massively parallel processor system - Cray T3D, Technical report 80922, Munchen, Germany, 1993.

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

[8] Kendall Square Research, KSR1 principles of operation manual, Kendall Square Research Corporation, Boston, MA, 1992.

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