

Computation-Communication Overlap on Network-of-Workstation Multiprocessors*

Gary Liu and Tarek S. Abdelrahman
Department of Electrical and Computer Engineering
The University of Toronto
Toronto, Ontario, Canada M5S 3G4

Abstract

This paper describes and evaluates a compiler transformation that improves the performance of parallel programs on Network-of-Workstation (NOW) shared-memory multiprocessors. The transformation overlaps the communication time resulting from non-local memory accesses with the computation time in parallel loops to effectively hide the latency of the remote accesses. The transformation peels from a parallel loop iterations that access remote data and re-schedules them after the execution of iterations that access only local data (local-only iterations). Asynchronous prefetching of remote data is used to overlap non-local access latency with the execution of local-only iterations. Experimental evaluation of the transformation on a NOW multiprocessor indicates that it is generally effective in improving parallel execution time (up to 1.9 times). The extent of the benefit is determined by three factors: the size of local-only computations, the significance of remote memory access latency, and the position of the iterations that access remote data in a parallel loop.

1 Introduction

Shared-memory multiprocessors built from Networks of Workstations (NOWs) are becoming increasingly popular as platforms for high-performance computing. In these systems, general-purpose workstations are connected together using a fast communication network, such as ATM or Fast Ethernet. Shared memory is supported using a Distributed Shared Memory (DSM) software, which makes the memory address space of each workstation part of a larger global address space. Hence, a process on one workstation can access data residing in this global address space, irrespective of the local address space in which the data actually

resides. Examples of such systems include the Stanford Distributed FLASH [10], the SUN s3.mp architecture [11] and the POW system at the University of Toronto [12].

Although NOW multiprocessors support shared memory, data transfers and synchronization are realized using underlying message passing mechanisms. This results in high memory access latencies when data being accessed is not resident in the local memory of a workstation. Indeed, the latency of a non-local memory access can be orders of magnitude higher than the latency of a local memory access. Programs that makes a large number of non-local memory accesses are likely to exhibit poor performance. Consequently, it is imperative that the latency of non-local memory accesses be reduced.

In this paper, we describe and experimentally evaluate a compiler transformation for parallel loops in scientific programs on NOW shared-memory multiprocessors. The transformation utilizes data distribution directives provided by programmers, or determined by the compiler, to identify the iterations of a parallel loop that require non-local memory accesses. These iterations are peeled from the parallel loop, leaving iterations that only access local data (called *local-only iterations*). The peeled iterations are scheduled for execution after the local-only iterations, and asynchronous reads (in the form of prefetches) are inserted before the local-only iterations. This effectively overlaps the latency of the remote accesses with the computation time of local-only iterations, reducing the impact of the latency on performance. We refer to this transformation as Computation-Communication Overlap, or CCO.

Experimental evaluation of the CCO transformation using representative applications on a shared-memory NOW multiprocessor indicates that the transformation is generally effective in hid-

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

ing the latency of remote accesses, and in improving parallel execution time. The performance of target parallel loops improves by a factor of up to 2.9; overall performance improves by a factor of up to 1.9. The extent of the benefit of the transformation is determined by three factors: (1) the size of the computation of the local-only iterations; (2) the significance of non-local memory access latency; and (3) the location in a parallel loop of the iterations that access remote data.

The remainder of this paper is organized as follows. Section 2 gives some background material on dependence and data distribution. Section 3 describes the computation-communication overlap transformation in details. Section 4 presents and discusses the results of our experiments. Section 5 reviews related work. Finally, Section 6 gives concluding remarks.

2 Background

The presentation of the CCO transformation requires familiarity with data dependence [15] and data distribution [7]. These topics are briefly described in this section.

2.1 Data Dependence

In a loop nest L , two statements S_1 and S_2 that reference a common array a are said to have a data dependence $S_1(\vec{i})\delta S_2(\vec{i}')$ if S_1 references the same array element in some iteration \vec{i} as S_2 in some iteration \vec{i}' , where iteration \vec{i} is executed before \vec{i}' . The dependence is called a *flow* dependence when S_1 writes a and S_2 reads a ; an *antidependence* when S_1 reads and S_2 writes; or an *output* dependence when both S_1 and S_2 write. S_1 is called the *source* of the dependence and S_2 is called the *sink* of the dependence. For each dependence $S_1(\vec{i})\delta S_2(\vec{i}')$, the dependence distance vector is defined as $\vec{d} = \vec{i}' - \vec{i}$. A dependence is said to be *loop independent* if its source and sink statements are in the same iteration, i.e., if $\vec{d} = 0$, otherwise it is *carried* by a loop. A loop is said to be *parallel* if it carries no dependence; otherwise the loop is said to be *sequential* or *serial*.

2.2 Data Distribution

Data distribution is an approach used by compilers for distributed memory multiprocessors, such as High-Performance Fortran (HPF) [9], to map array data onto separate address spaces. Although

such partitioning of an array is not necessary in the presence of shared-memory, it can be used to enhance memory locality—the compiler can place an array partition in the physical memory of the processor that uses it the most [1]. Data distribution can be specified by the programmer using compiler directives, as in Fortran D [7] and HPF [9], or automatically derived by the compiler [13]. In either case, we assume that directives in the input program specify data distribution.

An array is distributed by specifying a partitioning scheme for the array and by specifying a processor geometry to which the 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 $\mathbf{V}(V_0, V_1, \dots, V_{n-1})$, where V_i is the number of processors in the i^{th} dimension of the grid, and $V_0 \times V_1 \times \dots \times V_{n-1} = P$, 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 of the array. The DISTRIBUTE directive also maps the resulting array partitions onto a processor geometry declared using the PROCESSOR directive. 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 is called the *block-cyclic factor*. Finally, the * attribute is used to indicate that the corresponding dimension of the array is not distributed.

The use of the partitioning attributes in the dimensions of a multi-dimensional array results in useful data distribution schemes, which is illustrated in Figure 1 using a 2-dimensional array A. The processor geometry on which the array is mapped determines the number of processors assigned to each distributed dimension of the array.

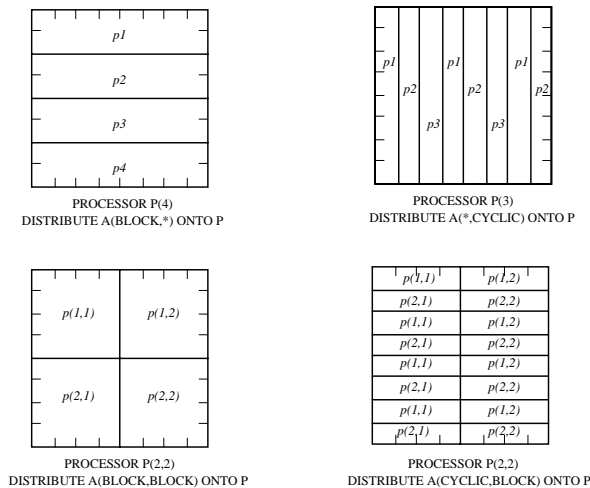


Figure 1: Data distribution examples.

3 The CCO Transformation

3.1 Overview

The goal of the CCO transformation is to overlap the latency caused by accesses to remote data in a parallel loop with the computations of the loop, effectively hiding remote access latency. This is accomplished by peeling iterations that read¹ remote data in the parallel loop, and re-scheduling these iterations for execution after the iterations that read only local data (referred to as the *local-only iterations*). Asynchronous reads in the form of prefetch instructions are initiated before the local-only iterations to overlap the latency of these remote reads with local computations, effectively hiding the latency.

The transformation is illustrated using the short program segment shown in Figure 2(a). The outer loop with the index j is parallel. The arrays A and B are partitioned using $(*, \text{BLOCK})$, i.e., in column blocks as shown in Figure 2(b). The code generated by the compiler for execution on processor 2 is shown in Figure 2(c). Iterations 26 and 50 of loop j in this code compute elements on the boundaries of the local data partition of array A residing in processor 2. These two iterations cause reads to elements of array B that are both local as well as remote. In contrast, iterations 27 to 49 read only local elements, and hence constitute the local-only iterations. Consequently, iterations 26 and 50 are peeled of the loop and rescheduled for execution after local-only iterations, as shown in Figure 2(d).

¹The use of data distributions and the owner-computes rule (described in the next section) dictates that all writes to array elements be local. Hence, the only source of remote accesses are reads.

The prefetch instructions are used to initiate remote memory reads to make needed elements of B residing in the memory modules of processors 1 and 3 into the local memory of processor 2.

There are 2 types of analyses that are used by the CCO transformation: (1) data distribution analysis, which is used to determine for a given processor which array elements are local and which elements are remote; and (2) communication analysis, which is used to determine which iterations of a parallel loop read only local array elements and which iterations read remote data. The transformation is then implemented using loop peeling and by inserting appropriate calls to prefetch instructions.

3.2 Data Distribution Analysis

This analysis uses the array distribution directives in the input program to map array partitions to processors and to partition parallel loops, i.e., determine the subset of loop iterations that will be executed by each processor. This analysis is largely based on that of Fortran D [7], and hence, will only be overviewed.

The mapping of data partitions to 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 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 $0, 1, \dots, P - 1$. Virtual processor $(v_0, v_1, \dots, v_{n-1})$ 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.

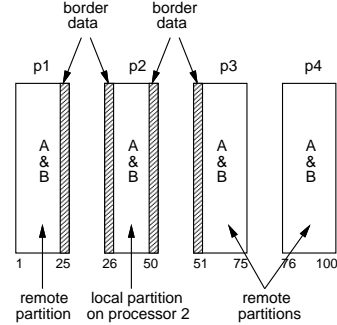
The owner-compute rule [7] is used to partition parallel loops. This rule specifies that a processor executes a statement that *computes* an array element (i.e., writes to it) if and only if it *owns* the array element, i.e., has the element in its local memory. Consequently, the subset of loop iterations 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 indices of array elements that reside in the local memory of a processor. It is calculated using the array distribution directives provided by the programmer. The *Global Iteration*

```

PROCESSOR P(4)
DIMENSION A(100,100)
DIMENSION B(100,100)
DISTRIBUTE A(*,BLOCK) ONTO P
DISTRIBUTE B(*,BLOCK) ONTO P
:
:
DOALL j=2,99,1
DO i=2,99,1
A(i,j)=B(i,j-1)+B(i,j+1)

```

(a) Program segment



(b) Data partitions

```

DO j=26,50,1
DO i=2,99,1
A(i,j)=B(i,j-1)+B(i,j+1)

```

(c) Code on processor 2

```

prefetch B(2:99,25)
prefetch B(2:99,51)

DO j=27,49,1
DO i=2,99,1
A(i,j)=B(i,j-1)+B(i,j+1)
DO i=2,99,1
A(i,26)=B(i,25)+B(i,27)
A(i,50)=B(i,49)+B(i,51)

```

(d) Peeling and inserting prefetchs

Figure 2: Overview of the CCO transformation.

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 Set with the set that describes the loop iteration space. The data distribution analysis is illustrated in Figure 3, which shows a simple example and the various index sets generated on a processor. The index sets are expressed using regular section descriptors, or RSDs [3]. 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.

3.3 Communication Analysis

This analysis 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 statements 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 for the array 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 index sets generated by communication analysis in the above example are also illustrated in Figure 3.

3.4 Prefetch Insertion

The compiler inserts a call to a prefetch routine to initiate asynchronous transfer of data from remote to local memory. This is done whenever the Remote Index Set for an array reference in a parallel loop nest is non-empty. The call to the prefetch routine are inserted immediately preceding the outermost parallel loop in the nest. The RSD representation of the Remote Index Set of each array is passed as a parameter to the prefetch routine. The routine uses the RSDs and a table containing the starting address of each array (initialized at run-time) to generate the addresses of the pages that must be prefetched to transfer remote data.²

²The prefetch supported by our DSM software is page-based; see [5] for details.

```

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

```

(a) A parallel loop code segment.

```

Local Index sets of a & 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: none }

```

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

Figure 3: Analysis for a simple loop nest.

3.5 Loop Peeling

The peeling transformation itself is always legal since only parallel loops are considered. A parallel loop has no loop carried dependences and hence its iterations can be executed in any order. The iterations that access remote data are removed from the loop and re-scheduled after the iterations that access only local data.

However, in some loop nests, the index of a serial loop is used to access an array dimension that is distributed. The dependences that are carried by the serial loop become *cross-processor* and force the serial loop to be executed in a pipelined “wave-front” fashion [7]. In this case, reordering the iterations of the loop is not legal because of the dependences. Consequently, loop peeling is not applied in the presence of cross-processor dependences.

It is important to note that it is possible to only insert the prefetch instruction to read remote data without peeling and re-scheduling loop iterations that access non-local data. In this case, the benefit of the CCO transformation depends on the location of the iterations that access non-local data in the parallel loop. If such iterations are towards the end of a parallel loop, the computations of the parallel

loop provide the necessary overlap. However, if the iterations are towards the beginning of the parallel loop, peeling is necessary to re-schedule the execution of the iterations after local-only computation and realize an overlap.

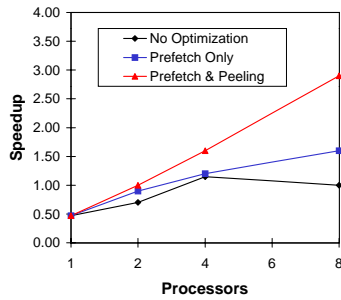
4 Experimental Evaluation

The CCO transformation was implemented in a prototype compiler called *Jasmine*, which is being developed at the University of Toronto. The compiler has four major phases: parallelism detection, cache locality enhancement, memory locality enhancement and code generation. The Polaris compiler [4] from the University of Illinois is used for parallelism detection. It has been extended to accept data distribution directives and to implement the CCO transformation as part of the memory locality enhancement phase.

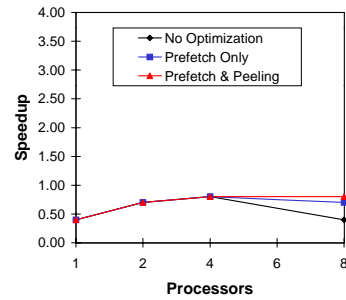
The compiler was used to automatically apply the transformation to 3 representative applications: the well known *Jacobi* kernel, red-black SOR (*rbSOR*) and the *swim* benchmark from *spec95*. The performance of the applications with and without the transformation was experimentally evaluated on the POW network-of-workstations multiprocessor [12], also developed at the University of Toronto. The POW multiprocessor consists of IBM RS/6000 workstations connected by Fast Ethernet. The experiments were conducted using 8 workstations. A version of *TreadMarks* [2] modified to support page-level prefetching [5] was used to provide cache-coherent shared-memory.

The speedup (with respect to sequential execution) of *Jacobi* is shown in Figures 4–6. A $(*, \text{BLOCK})$ data distribution is used for all arrays. In each figure, the speedup of only the target parallel loop targeted by the transformation as well as the speedup for the complete application are shown. In each case, 3 sets of results are presented. The first gives the speedup without any optimizations, and it is used for reference. The second set gives the speedup with only prefetches inserted, but without loop peeling. The final set gives the speedup after prefetches are inserted and loop peeling is applied.

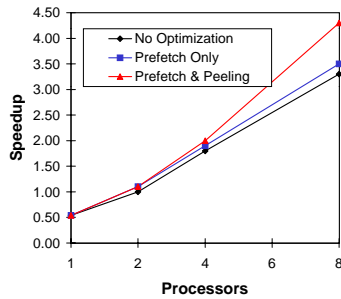
The speedup curves reflect that the performance of both the target parallel loop in *Jacobi* and of the overall *Jacobi* do benefit from the CCO transformation. The ratio of the parallel execution time at 8 processors with both prefetching and peeling applied to the parallel execution time (at 8 processors) with no optimization is shown in Table 1. The



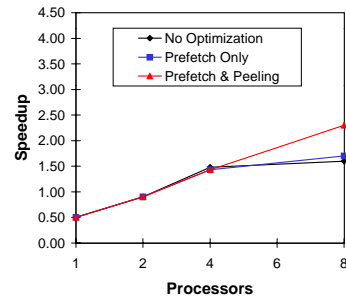
(a) Parallel Loop



(b) Application

Figure 4: Speedup of 256×256 Jacobi with 640 iterations.

(a) Parallel Loop



(b) Application

Figure 5: Speedup of 512×512 Jacobi with 640 iterations.

Table 1: Ratio of parallel execution times at 8 processor with and without transformation (640 iterations).

Problem Size	Loop Only	Application
256×256	2.9	1.9
512×512	1.3	1.3
1024×1024	1.1	1.1

table indicates that the performance of the parallel loop improves by up to 2.9 times. The performance of the overall kernel improves by up to 1.9 times.

The speedup curves also indicate the need for the peeling component of the transformation. Although there is benefit from just using prefetching, there is added benefit from combining peeling with prefetching. As discussed in Section 3.5, the prefetch instructions alone benefit performance when the iterations that access remote data are towards the end of the parallel loop. The loop computations provide sufficient overlap and there is little or no need to use the peeling component of the transformation to provide further overlap. However, when the iterations that access remote data are at the beginning of the parallel loop, there is no sufficient computation to overlap with the prefetches, and processors idle waiting for remote data. In this

case, peeling is necessary to delay the execution of these iterations. In the case of Jacobi, there are accesses to remote data both at the beginning and at the end of the parallel loop. Consequently, there is benefit to performance by using the prefetches alone and added benefit from peeling.

The speedup curves and Table 1 also indicate that the extent of the benefit of the transformation depends on the size of the data array and on the number of processors. The improvement in performance is higher when the input size is smaller. The transformed parallel program with both prefetching and peeling is 1.9 times faster than the original parallel program without optimization at 8 processors when a data size of 256×256 . However, the transformed parallel program is only 1.3 times faster when the data size is 512×512 , and only 1.1 times faster when the data size is 1024×1024 . This is because the size of the data determines the amount of local computation performed by each processor in the parallel loop. When the size of the data is relatively small (256×256), the amount of local computation in the parallel loop is small, and the impact of the remote memory accesses is more pronounced. Hiding the latency of remote accesses has a significant impact on performance. In contrast, when the size of the data is relatively

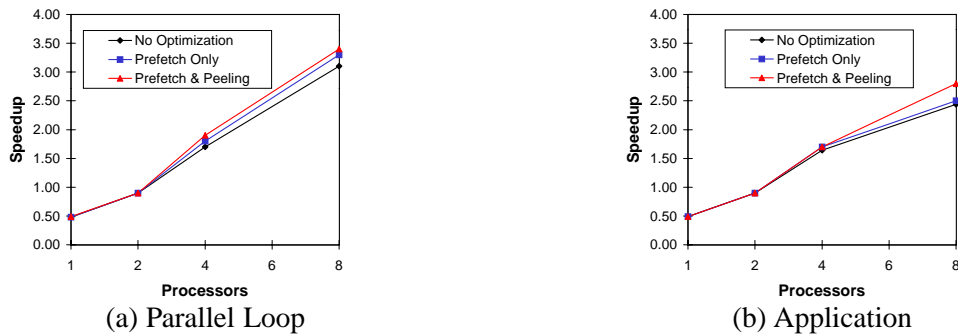


Figure 6: Speedup of 1024×1024 Jacobi with 640 iterations.

Table 2: Average ratio of un-successful prefetches at 8 processors (640 iterations).

Data Size	Average Ratio
128×128	6.03E-3
256×256	1.11E-3
512×512	0
1024×1024	0

large (1024×1024), the impact of remote memory accesses on performance is relatively small; the amount of local computation grows by $O(n^2)$ while the amount of non-local memory accesses grow by $O(n)$, where $n \times n$ is the problem size. Consequently, for larger data sizes there is less benefit to overall performance.

The computation time of the local-only iterations also determines the effectiveness of the CCO transformation in hiding the latency of remote accesses. Table 2 shows the average ratio of un-successful prefetches (i.e., prefetches that are issued by a processor, but are not complete by the time data is needed by the processor) to total prefetches for various data size of the Jacobi. When the size of the data is small, the amount of local-only computation is small and a fraction of prefetches are unsuccessful. However, when the size of the data is large, there is enough local-only computation to overlap with remote accesses and all of the prefetches that are issued succeed.

The speedup of `rbSOR` and of `swim` is shown in Figures 7 and 8, respectively. The magnitude of the performance improvement in each case that is less than that in the case of Jacobi. This is because of the large amount of local-only computation time compared to the time of non-local memory accesses. Furthermore, in the case of `swim`, the location of the iterations that access remote data is towards the end of the parallel loops, resulting in minimal benefit from peeling.

5 Related Work

There is a large body of work on the use of hardware and software prefetching to improve the performance of programs, on both single- and multi-processor systems (see [6]). These techniques differ from this work in two main aspects. First, the above techniques prefetch data into the *cache* of a processor. In contrast, the prefetch mechanism we use (developed by Chan [5]) brings data into the local memory of a processor, making the benefit of prefetching smaller. However, it avoids potential cache pollution problems. Second, the above techniques generally prefetch a single cache line. In contrast, the prefetch mechanism we use brings a whole page into memory, which makes it possible to implement prefetching with no hardware support [5].

There is also a large body of work on communication optimization in message passage programs that are compiler-generated using Fortran D or High-Performance Fortran [7, 8]. In particular, the CCO transformation is similar to *iteration reordering* described by Hiranandani, Kennedy and Tseng [8]. The CCO transformation can be considered as extending the applicability of iteration reordering to shared memory multiprocessors. However, there are some important differences. The use of prefetch instructions in a shared address space relieves the compiler from calculating “owners” of remote data and from using explicit send/receive instructions. On the other hand, while iteration reordering uses message passing to move exactly the needed data from remote to local memory, the prefetches move data in units of pages, which may contain un-needed data, and results in more overhead. In addition, the prefetches are non-binding and cannot be used for synchronization in a manner similar to message receives. Hence, if no sufficient

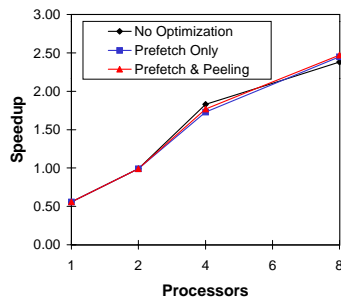


Figure 7: Speedup of rbsor.

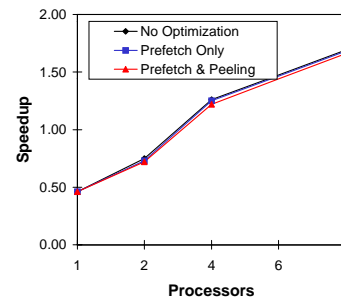


Figure 8: Speedup of swim.

local-only computation time exist, the full latency of remote accesses is incurred in addition to the overhead of prefetching.

6 Conclusions

This paper described and evaluated a transformation called CCO for hiding the latency of non-local memory access in parallel loops on shared-memory NOW multiprocessors. Experimental evaluation of the transformation indicates that it is effective in improving parallel performance. The benefit is highest when the impact of non-local memory accesses is high, and there is sufficient local computation to hide non-local memory accesses. The extent of the benefit also depends on the location in a parallel loop of the iterations that access remote data; the benefit is higher when such iterations are towards the beginning of the parallel loop.

References

- [1] T. Abdelrahman and T. Wong. Compiler support for array distribution on NUMA multiprocessors. *The Journal of Supercomputing*, to appear, 1998.
- [2] C. Amza, et al. TreadMarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.
- [3] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: the data access descriptor. *Journal of Parallel and Distributed Computing*, 9(6):154–170, 1990.
- [4] W. Blume, et al. Parallel programming with Polaris. *Computer*, 29(12):78–82, 1996.
- [5] C. Chan. Tolerating latency in software distributed memory systems through non-binding prefetching. M.Sc. Thesis, Dept. of Computer Science, Univ. of Toronto, 1997.
- [6] T. Chen and J. Baer. A performance study of software and hardware data prefetching schemes. In *Proc. of the Int'l Symp. on Computer Architecture*, pages 223–232, 1994.
- [7] S. Hiranandani, K. Kennedy and C. Tseng. Compiling Fortran D. *Comm. of the ACM*, 35(8):66–80, 1992.
- [8] S. Hiranandani, K. Kennedy and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. of the Int'l Conf. on Supercomputing*, pages 1–14, 1992.
- [9] C. Koebel, D. Loveman, R. Schreiber, G. Steele, Jr. and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.
- [10] J. Kuskin, et al. The Stanford Dash multiprocessor. In *Proc. of the Int'l Symp. on Computer Architecture*, pages 302–313, 1994.
- [11] A. Nowatzky et al. The s3.mp scalable memory multiprocessor. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 1–8, 1995.
- [12] The POW multiprocessor project. The Univ. of Toronto. A WWW document available at <http://www.eecg.toronto.edu/parallel/sigpow>, 1997.
- [13] S. Tandri and T. Abdelrahman. Automatic partitioning of data and computation on scalable shared memory multiprocessors. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 64–73, 1997.
- [14] B. Verghese, S. Devine, A. Gupta and M. Rosenblum. OS support for improving data locality on CC-NUMA compute servers. In *Proc. of ASPLOS*, pages 279–289, 1996.
- [15] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, Redwood City, CA, 1996.