Achieving Performance under OpenMP on ccNUMA and Software Distributed Shared Memory Systems *

B. Chapman, F. Bregier, A. Patil and A. Prabhakar
Department of Computer Science, University of Houston, Houston, TX

Abstract

OpenMP is emerging as a viable high-level programming model for shared memory parallel systems. It was conceived to enable easy, portable application development on this range of systems, and it has also been implemented on ccNUMA architectures. Unfortunately, it is hard to obtain high performance on the latter architecture, particularly when large numbers of threads are involved. In this paper, we discuss the difficulties faced when writing OpenMP programs for ccNUMA systems, and explain how the vendors have attempted to overcome them. We focus on one such system, the SGI Origin 2000, and perform a variety of experiments designed to illustrate the impact of the vendor's efforts. We compare codes written in a standard, loop-level parallel style under OpenMP with alternative versions written in an SPMD fashion, also realized via OpenMP, and show that the latter consistently provides superior performance. A carefully chosen set of language extensions can help us translate programs from the former style to the latter (or to compile directly, but in a similar manner). Syntax for these extensions can be borrowed from HPF, and some aspects of HPF compiler technology can help the translation process. It is our expectation that an extended language, if well compiled, would improve the attractiveness of OpenMP as a language for high performance computation on an important class of modern architectures.

Keywords: shared memory parallelism, parallel programming models, OpenMP, ccNUMA Architectures, restructuring, data locality, data distribution, Software Distributed Shared Memory

1 Introduction

Many modern supercomputers are cache-coherent Non-Uniform Memory Access architectures (cc-NUMAs). Such systems consist of a number of nodes, with memory and one or more processors, connected via a fast interconnect. Memory is globally addressed; code running on a processor may reference data stored at any location throughout the machine. Whenever remote data is accessed, the system fetches the data and, if necessary, invalidates copies in cache belonging to other processors. In many respects, these architectures appear to be

*This work was partially supported by NASA Ames Research Center under contract number ****, and by NSF under grant number NSF ACI 99-82160. Initial experiments were performed while three of the authors were in residence at ICASE, NASA Langley Research Center. These sources of support are gratefully acknowledged.

shared memory systems from a user's point of view. Their design overcomes the limitations of shared memory machines and very large ccNUMA systems have been constructed.

Such systems may typically be programmed as if they are distributed memory systems, under MPI or PVM, or as if they are shared memory systems. Given the fact that the system emulates shared memory, the latter programming model seems a natural approach for application development. But in contrast to true shared memory systems, the cost of accessing data at run time depends on its location relative to the processor that requires it. Distributed memory programming models force the user to code remote fetches explicitly and thus the application developer will take this aspect of the program's behavior into consideration. Shared memory programming models, by their very nature, do not provide for remote data fetches and thus a user need not be aware of their necessity. Worse, in such a paradigm the user has no control of the relative placement of data and computation, and thus no direct influence on the cost incurred by data references in a program.

In this paper, we discuss the use of the shared memory programming model OpenMP on ccNUMA architectures, and consider how to write scalable OpenMP applications despite this problem. We show a programming style in which the locality of data and code is taken into account. We also evaluate this programming style on a software distributed shared memory system (SDSM) in order to determine whether it can be expected to improve program performance on such systems also. Our results confirmed its usefulness for the system considered. Unfortunately, it is not an intuitive style, and more work is required of an application developer if code is converted to OpenMP under it. So one of the major benefits of OpenMP is partially lost in the process.

In recognition of the problems posed by OpenMP usage on ccNUMA architectures, hardware vendors have provided extensions to OpenMP to help a user manage the locality of data with respect to the work. The sets of extensions differ and thus code using them is no longer portable. We believe that a carefully considered, and modest, set of such extensions could be key to enabling users to obtain performance on ccNUMA architectures. They could be used to translate an intuitively developed OpenMP code to a higher-performing equivalent program, or could be directly compiled. In this paper we focus on the experiments performed and indicate the set of directives that we have begun to implement only briefly.

The remainder of the paper is organized as follows. In Section 2, we describe the characteristics of a ccNUMA system, and then briefly introduce the SGI Origin 2000, on which our experiments were performed, before discussing the OpenMP programming model and its usage. We explain the shortcomings of OpenMP as a programming model for such architectures, and outline the language features introduced by SGI and Compaq that are intended to overcome those shortcomings on their relative systems.

This is followed by a report in Section 3 on a set of simple OpenMP programming experiments carried out on a Silicon Graphics Origin 2000 at NCSA, some of which make use of SGI's OpenMP extensions. We subsequently show a second set of experiments that use an alternative OpenMP programming style in Section 4. Section 4.2 reports upon our experiences using this programming style on TreadMarks, an SDSM system from Rice University.

Finally, in Section 5 we briefly discuss a possible set of language extensions to OpenMP that might be used together with the straightforward "loop-level" programming style (as employed in the first set of experiments) and either translated to the alternative programming style, or compiled directly. We compare these with the features proposed by Compaq and

SGI, before discussing related work in Section 6 and reaching some conclusions in Section 7.

2 Programming ccNUMA Architectures

Several vendors market ccNUMA systems (Compaq, HP, SGI, Sun). These machines emulate shared memory while providing significantly higher levels of total performance. A typical ccNUMA platform is made up of a collection of Shared Memory Parallel (SMP) nodes, or modules, each of which has internal local shared memory; the individual memories together comprise the global memory. The entire memory is globally addressed, and thus accessible to all processors; however non-local memory accesses are more expensive than local ones. The memory hierarchy thus consists of one or more levels of cache associated with an individual processor, a node-local memory, and remote memory, main memory that is not physically located on the accessing node. A cache-coherent system assumes responsibility not only for fetching and storing remote data, but also for ensuring consistency among the copies of a data item. If data saved in a cache is updated by another processor, then the value in cache must be invalidated. Thus such systems behave as shared memory machines with respect to their cache management schemes.

2.1 The Silicon Graphics Origin 2000

The SGI Origin 2000 is such a system [15]. It is organized as a hypercube, where each node consists of a number of processors and memory that is local to them. The size of Origin systems is growing rapidly, and a 1024 processor machine is expected to appear soon.

Figure 1 shows the high level architecture of the Origin 2000. A node is formed by a pair of MIPS R12000 processors, connected through a hub, together with a portion of the shared memory. Multiple nodes are connected in a hypercube through a switch-based interconnect. One router is needed to connect up to 8 processors, since two pairs are connected directly via their hubs; two routers are needed to connect 16 processors, 4 for 32 processors and so on. Each MIPS R12000 processor has two levels of two-way set associative cache, where the first level of cache is on-chip and provides 32KB data cache and 32KB instruction cache (32-byte line size). The second level of cache is off-chip; it typically provides 1-4MB unified cache for both data and instructions (128-byte line size). All caches utilize a Least Recently Used (LRU) algorithm for cache line replacement. In addition, each node contains up to 4GB of main memory and its corresponding directory memory and has a connection to a portion of the I/O subsystem.

Page migration hardware moves data into memory close to a processor that frequently accesses it, thus increasing the data locality. The hub maintains cache coherence across processors using a directory-based invalidation protocol. While data only exists in either local or remote memory, copies of the data can exist in various processor caches. Keeping these copies consistent is the responsibility of the cache-coherent protocol of the hubs. The directory-based coherence removes the broadcast bottleneck that prevents scalability of the snoopy bus-based coherence. Latency of access to level 1 cache is approximately 5.5ns; for level 2 cache the latency is 10 times this amount. Latency of access to local memory is another 6 times as expensive, whereas latency to remote memory ranges from up to twice that for local memory, when at most 1 router is involved, to nearly 4 times the cost of access to local memory when 16 routers are configured. SGI reports that the bidirectional bandwidth is ca. 620 MBps for up to 3 routers (32 processors) and thereafter is ca. 310 MBps. However, the

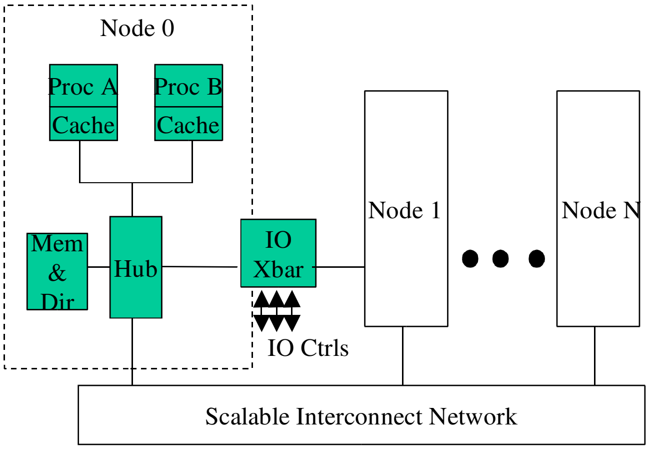


Figure 1. Architecture of the SGI Origin 2000

experienced cost of a remote memory access depends not only on the distance of its location, i.e. the number of hops required, but also on contention for bandwidth. Contention can have a severe impact on performance; it can arise as the result of many non-local references within a single code, or may be caused by the activities of other independent applications running on the same machine, and its effect is thus unpredictable.

The operating system supports data allocation at the granularity of a physical page (16KB). It attempts to allocate memory for a process on the same node on which it runs. However, results are not guaranteed [10]. Default strategies may be set by the user or the site. Typically, a default first-touch page allocation policy is used that aims to allocate a page from the local memory of the processor incurring the page-fault. In an optional round-robin data allocation policy, pages are allocated to each processor in a round-robin fashion. The Origin provides hardware and software support for page migration.

A ccNUMA platform may be programmed as a distributed memory or as a shared memory machine. Accordingly, a number of programming models are suitable for use on it. These include the de facto standards HPF [11], OpenMP [21], and MPI [18], as well as the hybrid MPI+OpenMP (e.g. [24]), shmem, and pthreads. We introduce the main features of OpenMP below, before discussing problems that arise when using it as a programming language for ccNUMA platforms.

2.2 OpenMP

OpenMP consists of a set of compiler directives, as well as library routines, for explicit shared memory parallel programming. OpenMP directives may be inserted into Fortran, C or C++ code in order to specify how the program's computations are to be distributed among the executing threads at run time. Based largely upon the earlier X3H5 standardization effort [30], it is a familiar programming model, enables relatively fast, incremental, application development, and has thus rapidly gained acceptance by users. The directives were developed by a consortium of vendors whose goal was to provide a portable programming interface for the growing number of SMP systems on the market. Compilers and tools for OpenMP are readily available.

The language extension is based upon the fork-join execution model, in which a single master thread begins execution and spawns worker threads to perform computations in parallel regions. It is the task of the user to specify parallel regions for execution by a team of threads. Directives are thus provided for declaring such regions, for sharing work among threads, and for synchronizing them. Worksharing directives spread loop iterations among threads, or divide the work into a set of parallel sections. Thus it is easy to specify task parallelism. Parallel regions may differ in the number of threads assigned to them at run time, and the assignation of work to threads may also be dynamically determined. It is thus relatively easy to adapt an OpenMP program to a fluctuating computational load, or even to a changing workload on the target platform. Users may set the number of executing threads; typically, there will be one thread per executing processor at run time.

The user is also responsible for detecting and dealing with race conditions in the code, as well as for thread coordination. Critical regions and atomic updates may be defined; variable updates may be ordered, or a set of statements executed by the first thread to reach them only. The OpenMP library implements locks.

Threads may have private copies of some of the program's data. OpenMP permits private variables, with potentially different values on each thread, within parallel regions. Fortran programs may have threadprivate common blocks, and the latest version permits private variables to be SAVEd also. Since OpenMP is a shared memory model, sequence and storage association is not affected by its directives.

OpenMP provides a fairly traditional, yet rich, set of features for parallel programming. It is particularly suitable for developing applications that will run on (true) SMPs, and experts have reported being able to port substantial codes in a matter of hours [3]. An innovative feature of OpenMP is the inclusion of so-called orphan directives, those that are dynamically within the scope of a parallel region. Their existence has greatly simplified the code migration process, since code must no longer be extracted from subprograms in order to specify worksharing. Moreover, the fact that programs may be incrementally migrated means that, in contrast to a global programming model such as MPI or HPF, a programmer need not consider the flow of control and data throughout an entire program while inserting directives. Given the combination of ease of development and the maintenance benefits of a high-level programming model, it is not surprising that OpenMP has rapidly been adopted.

2.3 OpenMP on the Origin 2000

OpenMP was developed as a high-level programming model for creating applications that are able to exploit the processing power of SMPs, and it is also implementable on ccNUMA systems. It promises to provide ease of development together with good performance on this range of architectures also.

SGI has provided an OpenMP implementation for the Origin 2000, and it is the programming model recommended by the vendor. The threads that execute a parallel region may run on processors across multiple nodes of the machine. When a loop is parallelized via a worksharing construct, for instance, loop iterations are executed on threads that may be located on different processors spread across the system, and the program data may also be spread across the different node memories. It is likely that a thread will reference data stored in non-local memory. But substantially different cost is involved in accessing the various levels of memory, as indicated above, and remote memory accesses can increase the memory latency by a significant factor; see [25] for a study of access delays related to memory hierarchy on the SGI Origin 2000. Further, if data is not stored where it is (mainly) used, network traffic may cause substantial bottlenecks.

In order to obtain high performance from ccNUMA systems, it is thus important that the placement of data and computations over the memories is such that the data accessed by each thread is largely local to the processor on which that thread is currently executing. OpenMP does not provide any mechanisms for controlling the placement of data relative to computation, so the vendor has supplied a number of mechanisms with which this may be influenced. The default strategies, in particular "first touch" data allocation, whereby data is allocated to memory local to the processor that first accesses it, were introduced above. This can be beneficial if used correctly; if the initialization is not parallelized, however, it may lead to the storage of large data structures on a single node. In such cases, the performance degradation can be significant. The operating system also provides for automatic page migration, whereby data that is remotely accessed may be transparently moved to memory near the processor that has referenced it. There are several problems with this in practice. First, the ability of the system to place data where it is used is limited by the amount of memory available; on a busy system in multi-user mode, it is possible that much of a program's data may be remotely stored. This has been studied in [10]. Worse, the operating system cannot know just when to move pages of data to be near the accessing processor. We turned the automation on when we first performed our experiments, and obtained very disappointing results.

Finally, SGI has provided extensions to OpenMP with which the user may influence the mapping of data and of threads explicitly. These are outlined below.

2.4 OpenMP Language Extensions for ccNUMA Platforms

The best solution to the problem of allocating data and threads in an OpenMP program would be to implement a transparent, and highly optimized, dynamic migration of data. However, there are several problems in providing such an option. As noted, is it very hard for the operating system to determine when to migrate data and current implementations do not perform well. Moreover, page-based storage is not always a suitable basis for an appropriate distribution of data. Both SGI and Compaq thus provide high level directives to specify data distribution and thread scheduling in OpenMP programs [5, 6, 27]. The extension sets differ. It is unfortunate that their syntax also differs, since there is substantial overlap in the core functionality of the two sets of directives.

A major component of both sets is the `DISTRIBUTE` directive. This specifies the manner in which a data object is mapped onto the system memories. Three distribution kinds, namely `BLOCK`, `CYCLIC` and `*`, are available to specify the distribution required for each dimension of an array. For example, in SGI FORTRAN the following statement will distribute the two dimensional array `A` by block in the first dimension:

```
!$SGI DISTRIBUTE A(BLOCK,*)
```

The first dimension is distributed page-wise in blocks over the local memories, while the second dimension is not distributed at all. An algorithm based upon the HPF block distribution strategy is used to determine which processor a *page* of memory is to be associated with. The entire page is allocated to the associated local memory, thus approximating an HPF-style block distribution. A `CYCLIC` distribution will assign pages of data elements to memories in round-robin fashion. The Compaq directive corresponding to the above directive is:

!DEC$ DISTRIBUTE A(BLOCK,*)

These directives influence the virtual memory page mapping of the data object, hence the granularity of distribution is limited by the granularity of the underlying pages, which is at least 16KB on the SGI Origin 2000. The advantage is that these directives can be added to an existing program without any restrictions, since they do not change the arrangement of the data object itself. A major disadvantage is that they are unsuitable for distributing small arrays. Both sets of extensions also provide a `REDISTRIBUTE` directive, with which an array distribution can be changed dynamically at run-time.

It is possible to perform data distribution at element granularity rather than page granularity. This involves rearranging the layout of the array in memory so that two elements which should be placed in different processor's memories are stored in separate pages, which may require padding the array or other arrangements. The resulting data layout may violate the standard language array layout specifications, but it guarantees the specified distribution at the element level. The data mappings are defined as in the HPF standard. The SGI FORTRAN directive for this is:

!$SGI DISTRIBUTE_RESHAPE A(BLOCK,*)

Compaq requires that the `NOSEQUENCE` directive be supplied along with the `DISTRIBUTE` directive in order to specify element granularity. There are many limitations on the use of elementwise distributed arrays [27], as a result of the non-standard layout in memory.

Both vendors also supply directives to associate computations with the location of data in storage. Compaq provides the `NUMA` directive to indicate that the iterations of the immediately following `PARALLEL DO` loop are to be assigned to threads in a NUMA-aware manner, i.e. according to the distribution of the data. It may distribute multiple levels of the loop, in contrast to the single level specified by the OpenMP standard. The `ON HOME` directive informs the compiler exactly how to distribute iterations over memories, `ALIGN` is used for specifying alignment of data, `MEMORIES` is roughly equivalent to the HPF `PROCESSORS` directive, although it maps data to local memories rather than individual processors, and the `TEMPLATE` directive is used to define a virtual array. SGI similarly provides `AFFINITY`, a directive that can be used to specify the distribution of loop iterations based on either `DATA` or `THREAD` affinity (thus closely related to the `ON HOME` directive of Compaq) and the `NEST` directive, corresponding to Compaq's `NUMA`.

In addition to these, both vendors have added lower-level directives to directly influence the location of pages in memory. The `PAGE_PLACE` directive can be used to dynamically change the virtual memory mapping of an address range, so that the pages are allocated from a particular node's memory. The syntax for this directive is:

!$SGI PAGE_PLACE (start_address, size, threadnum)

The pages are mapped to the memory of the node executing thread `threadnum`.
Similarly, Compaq provides `MIGRATE_NEXT_TOUCH` and `PLACE_NEXT_TOUCH` directives to dynamically enable a new virtual memory mapping, where the former moves old data values to the new location and the latter does not. A `MIGRATE_TO_OMP_THREAD` causes a set of pages to be placed on the node where the specified thread is executing.

3 Experiments with OpenMP on an Origin 2000

The experiments described in this section were conducted on the Origin 2000 systems at the National Center for Supercomputing Applications (NCSA) in multiuser mode. SGI's MIPSpro7 Fortran 90 compiler was used with the following options:

- -mp : Generates multiprocessing code.
- -64 : Generates 64-bit objects (default on NCSA origin)
- -mips4 : Generates code using the MIPS4 instruction set (default at NCSA)
- -r1000 : Schedules code for the r10000 chip and uses system libraries specific to the r10000 if available(default on NCSA Origin)
- -Ofast : uses optimizations selected to maximize performance for target platform.
- -O3(aggressive optimization) and -IPA(interprocedural analysis).

The most suitable "first touch" default strategies were used in each case.

The experiments were initially performed with the page migration switched to its default mode on the NCSA platforms, in which the system attempts to migrate data to the referencing processors. The resulting performance figures were consistently much poorer than those obtained without it. For the results shown below, the Origin's `_DSM_MIGRATION` (page migration) environment variable was set to OFF. However, we also provide results for the Jacobi stencil with page migration turned on, so that the reader may compare them with other performance figures for this code.

The speedup figures shown in this and the following section have all been normalized. They are speedups with respect to the serial time of the initial OpenMP version (without vendor-specific directives), compiled without the multiprocessing code option. We have not included results for more than 64 (Jacobi) and 32 (LU, LBE) processors respectively, since for the matrix sizes used to test these results, there was not enough computation remaining to keep additional threads busy.

3.1 A Jacobi Stencil

The Jacobi method is one of the simplest numerical solvers for partial differential equations. Although it converges very slowly, it exhibits excellent spatial locality and is thus sometimes used in parallel computations. In this short code, almost all data accesses can be made local (see Prog. 1 and Fig. 2 (a)). Thus it should be possible for a system to obtain good performance on a Jacobi stencil under OpenMP without user-specified distribution directives. We may exploit the Origin's default first touch page allocation policy to ensure that pages are distributed among the executing processors. There will be shared read access to the boundary columns. The use of SGI's `DISTRIBUTE (*,BLOCK)` directive will do exactly the same thing, the only difference being that it can be set up at compile time.

The performance figures shown are based on runs with a 1024 by 1024 matrix of double precision data. The size was chosen so that the data fits precisely into a number of pages for all the numbers of threads used, although the system does not guarantee that data will start on a page boundary. Only the elementwise `DISTRIBUTE_RESHAPE` directive will guarantee

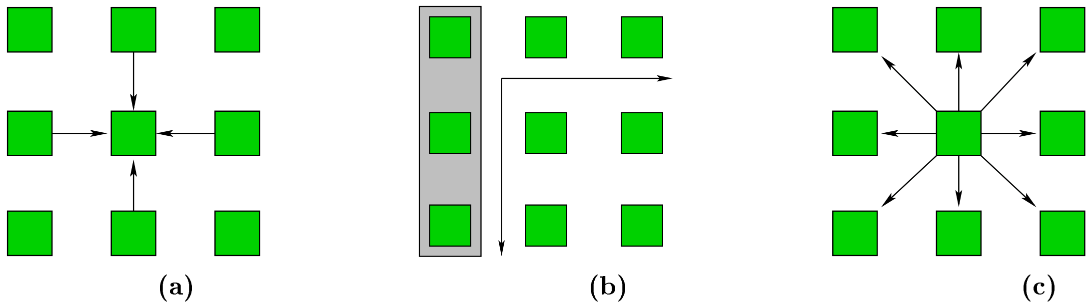


Figure 2. Jacobi (a), LU (b) and LBE (c) access stencils

```
!$OMP PARALLEL DO
  do j = 1,n
     do i = 1,n
        A(i,j) = (B(i-1,j)+B(i+1,j)+B(i,j-1)+B(i,j+1)) * c
     end do
  end do
!$OMP END PARALLEL DO
```

Program 1. Simple OpenMP version of Jacobi Kernel

this. All but the first and last threads must read 2 columns that are updated by another thread. If we ensure that the updating thread is the first to reference the data, the first touch policy will realize a pagewise block distribution of the second dimension. Threads will need to read 2 * 1024 elements (two half-pages, or 128 cache lines) that are stored in proximity to another thread at each iteration. But no matrix elements are updated by more than one thread.

We first show results obtained with the `_DSM_MIGRATION` switch set on (Fig. 3 (a)). In this case, we obtain close to linear speedup with 8 and fewer processors, but after that, it drops off sharply. If we compare this with our other results (Fig. 3 (b)), it is clear that pages are indeed being migrated; they will also presumably migrate back to their previous location, where they are updated in each iteration. With an increasing number of processors, more and more pages are moved. When more than 8 processors are involved, some of them will move across two hubs, and from 16 processors onward, across one or more routers. Network contention will increase with the additional volume.

Assuming that data is allocated locally to the thread that updates it, when executed with

```
!$SGI DISTRIBUTE A(*,block), B(*,block)
!$OMP PARALLEL DO
  do j = 1,n
     do i = 1,n
        A(i,j) = (B(i-1,j)+B(i+1,j)+B(i,j-1)+B(i,j+1)) * c
     end do
  end do
!$OMP END PARALLEL DO
```

Program 2. Jacobi Kernel Jacobi with `DISTRIBUTE`

```
!$SGI DISTRIBUTE_RESHAPE A(*,block), B(*,block)
!$OMP PARALLEL DO
  do j = 1,n
     do i = 1,n
        A(i,j) = (B(i-1,j)+B(i+1,j)+B(i,j-1)+B(i,j+1)) * c
     end do
  end do
!$OMP END PARALLEL DO
```

Program 3. Jacobi Kernel with `DISTRIBUTE_RESHAPE`

two threads. each thread will have 256 pages of data. For sixteen, each will have 32 pages. Data will fit into level 2 cache from 4 threads onward. So it is not surprising that superlinear speedup is observed at this point (see Fig. 3 (a)).

The first touch policy (Prog. 1) and `DISTRIBUTE (*,BLOCK)` (Prog. 2) realize the same data mapping, and since, in this example, each array element will be updated by only one processor, the page mapping remains the same after initialization in both cases. The compile time mapping enabled by the user directive provides better performance when the number of processors is increased. Note that we were only able to achieve the benefit of the `DISTRIBUTE_RESHAPE (*,BLOCK)` directive when using the *fast* compiler switch. Without it, performance was poor since the compiler did not optimize the pointer arithmetic introduced to realize accesses to the array elements in this version. Performance figures are consistently good for all three versions of the Jacobi code.

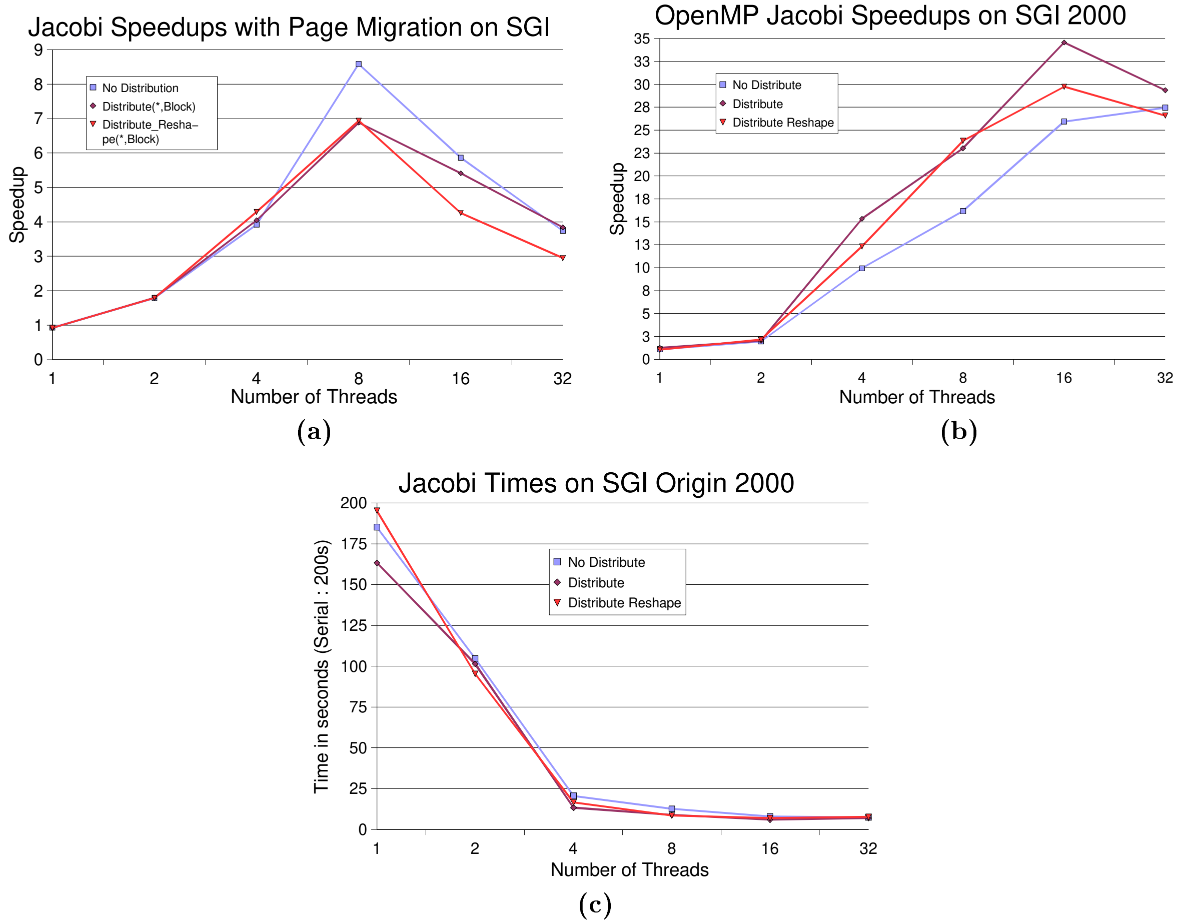


Figure 3. OpenMP Jacobi speedups on SGI O2000 with page migration (a) without page migration (b), and execution times without page migration (c)

3.2 LU

Our second experiment performed an LU factorization, another standard numerical solution method, in which a matrix is factorized into upper and lower triangular matrices. As with the Jacobi code, our program only requires shared read access, and that only on one column per iteration. However, the column involved differs in each iteration. The simple version used here does not implement pivoting (see Prog. 4).

The matrix size operated on reduces progressively (see Fig. 2 (b)). Therefore, we ensure that all threads continue to participate in the computation and ensure good load balancing by using a `CYCLIC` distribution. This can be achieved using the page-based `DISTRIBUTE` directive (Prog. 5) or the elementwise `DISTRIBUTE_RESHAPE` (Prog. 6), together with the `CYCLIC` data distribution in the second dimension. The `AFFINITY` clause is needed in order to ensure that the `DO` loop is distributed according to the data mapping. Otherwise, the default mapping of loop iterations to threads would be applied, and that would assign iterations to threads by block. Timings for the code without the `AFFINITY` directive were approximately 40 % higher than those shown here.

We show results for our LU computation based upon runs with a 2048 by 2048 real matrix in Fig. 4. For the version without distribution directives, the speedup is less than

```
!$OMP PARALLEL
  do k = 1,n-1
!$OMP SINGLE
     lu(k+1:n,k) = lu(k+1:n,k)/lu(k,k)
!$OMP END SINGLE
!$OMP DO
     do j = k+1, n
        lu(k+1:n,j) = lu(k+1:n,j) - lu(k,j) * lu(k+1:n,k)
     end do
!$OMP END DO
  end do
!$OMP END PARALLEL
```

Program 4. Simple OpenMP versions of LU

```
!$SGI DISTRIBUTE lu(*,CYCLIC)
!$OMP PARALLEL
  do k = 1,n-1
!$OMP SINGLE
     lu(k+1:n,k) = lu(k+1:n,k)/lu(k,k)
!$OMP END SINGLE
!$OMP DO
     do j = k+1, n
        lu(k+1:n,j) = lu(k+1:n,j) - lu(k,j) * lu(k+1:n,k)
     end do
!$OMP END DO
  end do
!$OMP END PARALLEL
```

Program 5. LU with `DISTRIBUTE`

linear after 16 processors are used. The other two versions continue to have nearly linear speedup on 32 processors. In the former case, the initial data mapping will lead to an increasing load imbalance, as well as to larger numbers of remote fetches as the computation progresses. The `DISTRIBUTE` and `DISTRIBUTE_RESHAPE` versions perform somewhat better, since the distribution reflects the load-balancing requirements.

```
!$SGI DISTRIBUTE_RESHAPE lu(*,CYCLIC)
!$OMP PARALLEL
  do k = 1,n-1
!$OMP SINGLE
     lu(k+1:n,k) = lu(k+1:n,k)/lu(k,k)
!$OMP END SINGLE
!$OMP DO
     do j = k+1, n
        lu(k+1:n,j) = lu(k+1:n,j) - lu(k,j) * lu(k+1:n,k)
     end do
!$OMP END DO
  end do
!$OMP END PARALLEL
```

Program 6. LU with `DISTRIBUTE_RESHAPE`

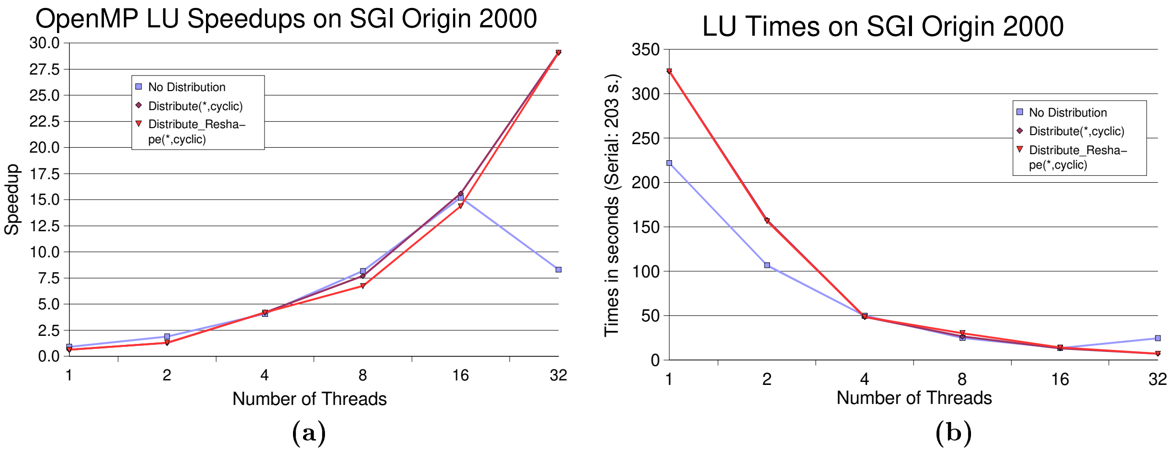
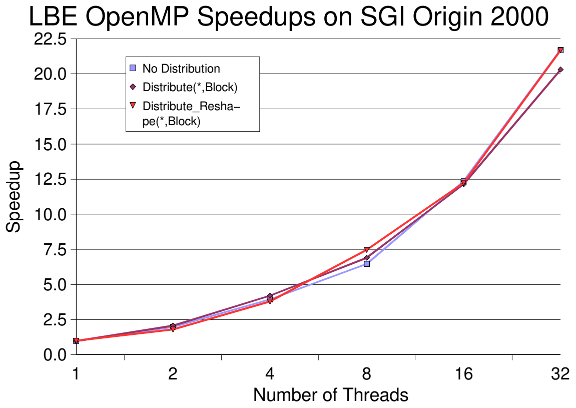


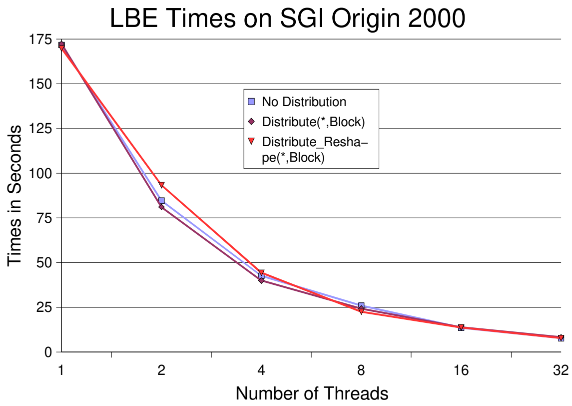
Figure 4. OpenMP LU speedups (a) and times (b) on SGI O2000

3.3 LBE

Our third experiment makes use of LBE, a computational fluid dynamics code that solves the Lattice Boltzmann equation. It was provided by Li-Shi Luo of ICASE, NASA Langley Research Center [31]. The numerical solver employed by this code uses a 9-point stencil. However, unlike the Jacobi solver, the neighboring elements are updated at each iteration (cf. Fig. 2 (c)). The `Collision_advection_interior` subroutine with a write shared data access is shown in Program 7. The shared memory policy permits multiple reads on the same cache line (or page) but not multiple writes. Thus LBE allows us to analyze this weakness of ccNUMA architectures. We developed three versions of this code just as in the previous cases. In the versions with distribution directives, the matrices are `BLOCK` distributed in the last dimension to ensure good data locality and an even load balance.

The experiments were performed with a 256 by 256 matrix (see Fig. 5). Here all three versions behaved similarly up to 32 processors (Prog. 9). Speedups are lower than those previously reported, a result of the fact that each processor has to access the (logical) neighbor's memory for writing rather than reading as in the case of Jacobi or LU. Cache lines at the boundaries of local data will ping-pong between a pair of processors where the corresponding





(a) **(b)**

Figure 5. OpenMP LBE speedups (a) and times (b) on SGI O2000

```
!$OMP PARALLEL
  do iter = 1, niters
     Calculate_ux_uy_p
     Collision_advection_interior
     Collision_advection_boundary
  end do
!$OMP END PARALLEL
```

```
Collision_advection_interior:
!$OMP DO
  do j = 2, Ygrid-1
     do i = 2, Xgrid-1
        f(i,0,j)   = Fn(fold(i,0,j))
        f(i+1,1,j) = Fn(fold(i,1,j))
        f(i,2,j+1) = Fn(fold(i,2,j))
        f(i-1,3,j) = Fn(fold(i,3,j))
        .....
        f(i+1,8,j-1) = Fn(fold(i,8,j)
     end do
  end do
!$OMP END DO
```

Program 7. OpenMP version of the LBE Algorithm and Collision_advection_interior Kernel

threads execute. For 16 to 32 processors, remote data fetches might involve 2 hops through the network, and thereby longer remote access times may be observed.

```
!$SGI DISTRIBUTE ux(*,BLOCK),uy(*,BLOCK),p(*,BLOCK),f(*,*,BLOCK),fold(*,*,BLOCK)
!$OMP PARALLEL
  do iter = 1, niters
     Calculate_ux_uy_p
     Collision_advection_interior
     Collision_advection_boundary
  end do
!$OMP END PARALLEL
```

Program 8. LBE kernel with `DISTRIBUTE`

```
!$SGI DISTRIBUTE_RESHAPE ux(*,BLOCK),uy(*,BLOCK),p(*,BLOCK),f(*,*,BLOCK),fold(*,*,BLOCK)
!$OMP PARALLEL
  do iter = 1, niters
     Calculate_ux_uy_p
     Collision_advection_interior
     Collision_advection_boundary
  end do
!$OMP END PARALLEL
```

Program 9. LBE kernel with `DISTRIBUTE_RESHAPE`

4 An Alternative OpenMP Programming Style for ccNUMA Platforms

ALthough each of the above provided reasonable performance, it is possible to do better. It has been pointed out by several application developers that a programmer can explicitly take care of the deep memory hierarchy of the underlying architecture by writing an OpenMP program in so-called SPMD style. A strategy for doing so makes use of data privatization in order to store all data as close as possible to the processor that most frequently uses it, and to minimize sharing of data. We expected this style to have a considerable impact on the performance of our third code.

Under the SPMD strategy, we distribute the arrays equally among the processors and convert the local part into an array that is private to each thread. One or more shared buffers are created to exchange data as needed between the threads. For instance, if one processor must read a row that is stored in the private memory of the "neighbor" thread, a shared array with the size of a row is created as a buffer to copy in this data. Once data has been copied into the shared buffer, it may be written into an array that is private to the processor that needs it. This is most efficiently performed by extending the size of each private array to include the "shadow" regions, as is also realized via a `SHADOW` directive in HPF. The programmer must explicitly synchronize reading and writing of buffer data. Thus each thread works on its private data, and sharing is enabled through small shared buffers. The resulting code resembles an MPI program to some extent, but it is easier to specify and potentially provides better performance than the corresponding MPI code [29].

The SPMD style has been used in conjunction with OpenMP by other researchers, but more often codes have used a mixture of OpenMP and MPI for better portability (cf. [14, 28, 9, 7]). In [29], a comparison of SPMD OpenMP, MPI and Co-Array Fortran programming styles is based on an ocean model. The author points out that using halos (or shadows) and an SPMD style also reduces false sharing.

4.1 Privatized Code Versions on SGI

Jacobi in SPMD style: In order to create this version of our Jacobi code, we divide arrays `A` and `B` among the processors in the second dimension, and create pairs of buffers for exchanging data at the two boundaries (one each for the first and last column) (Fig. 6). The size of the second dimension of the private arrays `A` and `B` is now (*n + 2 /no. of threads*), where n is the original size, and space is reserved for the shadow area. The size of the first dimension remains the same as in the shared version. The shared buffers are of size (2*N-2)* *columnsize*, where N is the number of threads.

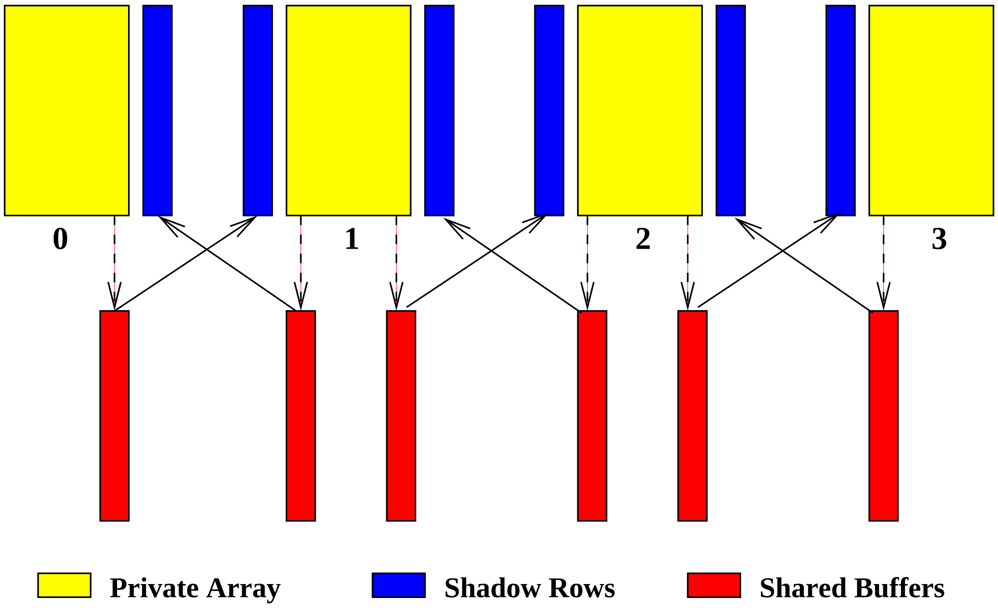


Figure 6. Jacobi SPMD stencil

At the start of the `PARALLEL` region (see Prog. 10), the previously defined buffers are declared to be shared, and `A` and `B` are declared to be private to each thread.

First, a thread copies its first and last column into the appropriate columns of the shared buffers to initialize them. This is followed by an OpenMP `BARRIER` so that no thread accesses the buffers until they have been written. The threads then copy appropriate columns from the buffers into their shadow area. The rest of the Jacobi kernel remains the same as before, with each thread excuting the code to calculate its own portion of data.

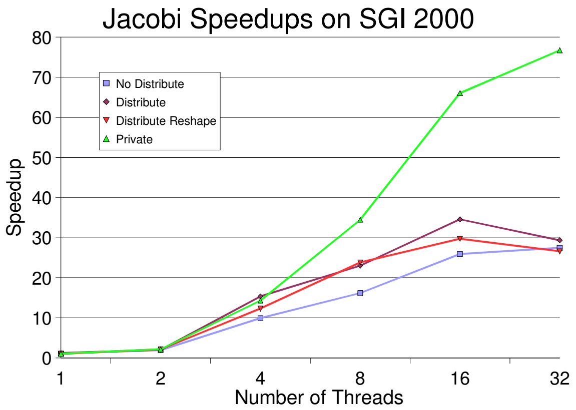
The only difficulty in constructing the SPMD version of the code is to properly synchronize writing to and reading from the shadow areas and buffer columns.

This version of the program shows superlinear speedup (Fig. 7 (a)). Not only is local cache used efficiently, there is less intrusion from the operating system while handling shared variables. Shared buffers are updated only once per iteration, and the update occurs separately from the rest of the computation. The SGI compiler is able to do a particularly good job of optimizing such "vector" updates, so that data transfer cost is further reduced. This good performance continues up to 128 processors, although speedup is no longer linear, as a

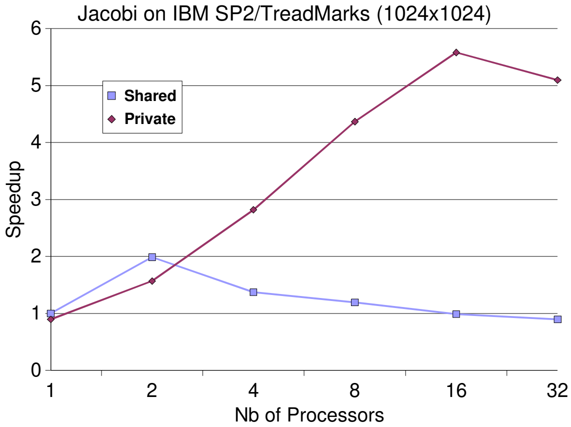
```
!$OMP PARALLEL SHARED(buflower,bufupper,chunk) &
!$OMP PRIVATE(A,B,thdno,noofthds,start_x,end_x,start_y,end_y,i,j,k)
      .......
      .......
   do k = 1, ITER
        bufupper(1:n,thdno) = A(1:n,chunk)
        buflower(1:n,thdno) = B(1:n,1)

!$OMP BARRIER

        B(1:n,0) = bufupper(1:n,thdno-1)
        B(1:n,chunk+1) = buflower(1:n,thdno+1)

        do j = start_y, end_y
          do i = start_x, end_x
            A(i, j) = (B(i-1, j) + B(i+1, j) + B(i, j-1) + B(i,j+1)) * c
          enddo
        enddo

        do j = start_y, end_y
           B(1:n, j) = A(1:n, j)
        enddo
   enddo
!$OMP END PARALLEL
```

Program 10. Jacobi Kernel in SPMD style

result of the decreasing computation per thread.



(a)



(b)

Figure 7. Jacobi speedups on SGI O2000 for SPMD version (a) and on IBM SP2 (b) with TreadMarks

LU in SPMD style: This code uses only one shared buffer of the size of one column to share the pivot column among the processors (Fig. 8). Each thread requires a private buffer of the same size. We realize the cyclic data distribution previously chosen by assigning columns to threads in a round robin fashion. The size of the second dimension will be n / *No. of threads*), where n is the original size of this dimension.

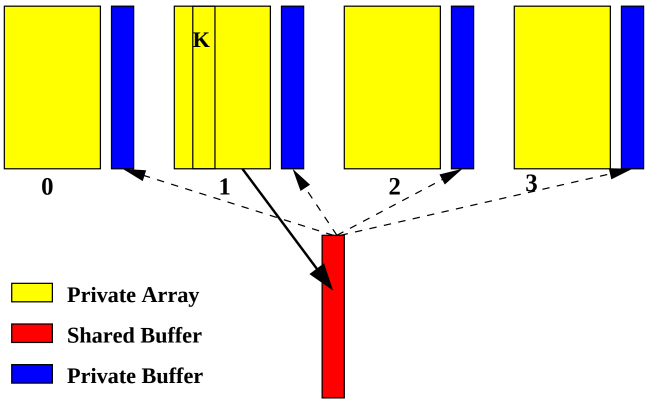
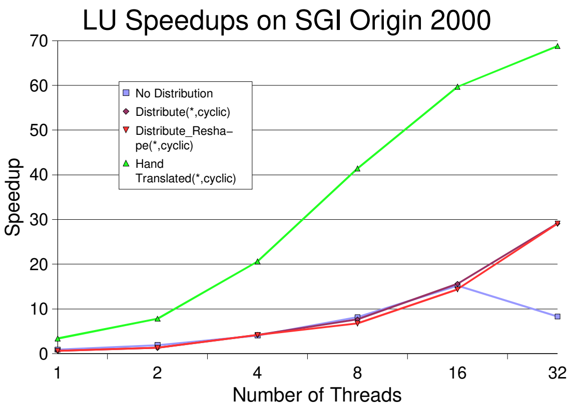


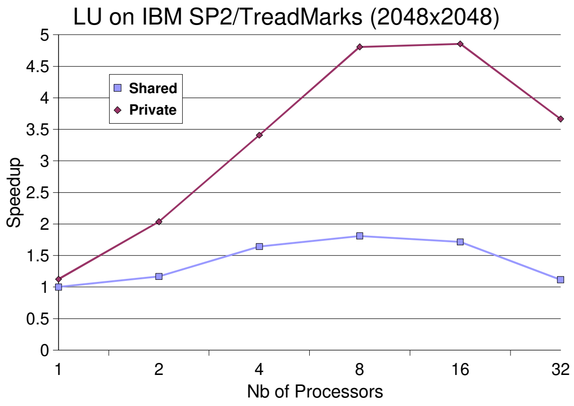
Figure 8. LU SPMD stencil

In the `PARALLEL` region (see Prog. 11), each column is chosen in turn and normalized. After normalizing, the pivot column is copied to the shared buffer. All other threads wait at the barrier. They then copy the contents of the shared buffer to their private buffer. The rest of the computation in the kernel involves only the columns to the right of the normalized one, and each thread can perform its portion independently on its own data. Under the cyclic distribution, selecting the right column in the right thread at each iteration is the only cumbersome task when creating the code.

The only sequential part of this version is when the pivot column is copied by the thread that "owns" it to the shared buffer, while the other threads wait at the barrier. The rest of the computation is completely parallel, with each thread working on its private array. The super linear speedup (Fig. 9 (a)) can be explained by the cache effect and efficient handling of the shared buffer updates.



(a)



(b)

Figure 9. LU speedups on SGI O2000 (a) and IBM SP2 (b)

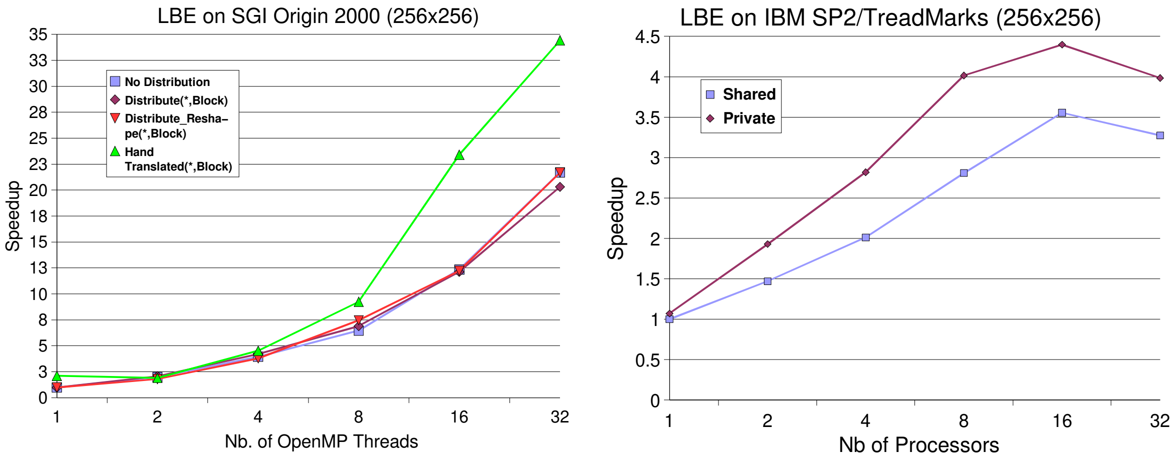
```
!$OMP PARALLEL SHARED(shbuf)&
!$OMP PRIVATE(lu,sumP,counter,noofthds,id,i,j,K,pribuf,Npri)
   DO K=1,N-1

! Compute Column K
      if(mod(K-1,noofthds).eq.id) then
          lu(K+1:N,counter)  = lu(K+1:N,counter) / lu(K,counter)

!move the column K to the sharedbuffer
          shbuf(K+1:N) = lu(K+1:N,counter)
          counter = counter+1
      end if
!$OMP BARRIER

!move the sharedbuffer to private buffer or column 0
      pribuf(K+1:N) = shbuf(K+1:N)

! Update Right Part (Column J+1:N)
      do j = counter, Npri
         lu(K+1:N, j) = lu(K+1:N, j) - lu(K, j) * pribuf(K+1:N)
      end do

   END DO
!$OMP END PARALLEL
```

Program 11. LU Kernel in SPMD style

LBE in SPMD style: The SPMD version of LBE realizes a `BLOCK` distribution of the arrays `ux, uy, p, f, fold`. Only the array `f`, which has a shared write access, requires a shadow column on either side of the private data. In contrast to the previous codes, data is written by a neighboring thread and must be subsequently read by its "owner". After the shadow columns are updated in subroutine `Collision_advection_boundary`, their contents are copied to shared buffers. After the copy has completed, the owner of the data may copy it into its private array. The synchronization needed to do this is the only difficult part when creating the SPMD code.

```
!$OMP PARALLEL
  do iter = 1, niters
     Calculate_ux_uy_p
     Collision_advection_interior
     Collision_advection_boundary
     Synchronize
  end do
!$OMP END PARALLEL
```

Program 12. LBE Kernel in SPMD style



(a) (b)

Figure 10. LBE speedups on SGI O2000 (a) and IBM SP2 (b)

The SPMD version of LBE shows a remarkable increase in performance for 16 and 32 processors (Fig. 10 (a)). In this version, the processors work in parallel on their private data till they reach the end of the iteration. Previously, non-local updates, and the copying of cache lines at boundaries, occurred throughout the computation. Here, sharing is again restricted to the buffer arrays, and it occurs at specific points, in a manner that easily permits compiler optimization. The speedup is lower beyond 32 processors, as a result of the relatively small matrix size used in the experiment (256 by 256).

In all cases, the SPMD program versions exhibit better speedup than those obtained by using the SGI directives `DISTRIBUTE` or `DISTRIBUTE_RESHAPE`. The translation from the loop parallel OpenMP code to the SPMD OpenMP mode was straightforward in each case. The main task required was the selection of a distribution strategy for the program's data. Indeed, after computing the local size of data objects, including shadow regions, it is just a matter of introducing buffer copying and synchronization to ensure that data is read from buffers after it has been written to them.

4.2 Experiments in OpenMP Programming Styles on TreadMarks

TreadMarks [1, 13, 32] is a Software Distributed Shared Memory system (SDSM) that uses the operating system's virtual memory interface to implement the shared memory abstraction. It works on most UNIX systems that provide a mechanism for a user process to get memory violation notifications. To maintain memory consistency, TreadMarks employs an extension to the Release Consistency protocol (RC) [4] called the Lazy Release Consistency protocol [12].

RC is a relaxed memory consistency model that allows a processor to delay making its changes to the shared data visible to other processors until a certain synchronization point is reached. This reduces the overall number of messages that must be transferred and allows the messages to be grouped together with data transfer. Lazy Release Consistency (LRC) [12] is an extension of RC wherein the propagation of modifications is postponed until the time of the acquire. A release in LRC is a completely local event and requires no communication. However, at an acquire, the acquiring processor must determine which modifications it needs from which processors, according to the definition of RC. The lazy implementation aims at reducing the number of messages and the amount of data transferred.

To eliminate the effects of false sharing, TreadMarks uses the multiple-writer protocol. With this protocol, two or more accesses can simultaneously occur on the same page and the modifications are done on a local copy of the shared page. When the processors reach a synchronization point, they exchange the diffs created by comparing the original copy to the modified local copy. The processors then apply each other's diffs on the local copy. In TreadMarks, the diffs are created only when requested by a processor, and not at every release and acquire. This lazy diff creation helps reduce the overall number of diffs created and can improve the performance.

The structure of the TreadMarks system is similar in spirit to ccNUMA systems, and any technique that is expected to give better performance on a ccNUMA machine, is also expected to yield some performance gain for a SDSM system. Thus we wrote versions of the above applications for TreadMarks using two approaches, one called *shared* where the full arrays are shared, as in OpenMP without any data distribution, and another one called *private* where the arrays are privatized to realize the SPMD style of OpenMP (as in the hand-coded versions presented in Section 4.1). Both kinds of code are very similar to the corresponding source codes written for the Origin (see, for example, the LU code in Prog. 13 for TreadMarks and in Prog. 11 for SPMD OpenMP).

The results we obtained are given in Figures 7 (b) for Jacobi, 9 (b) for LU and 10 (b) for LBE. All the test runs were done on the IBM SP2 system at the University of Houston, on thin nodes with 128MB memory and running at 120MHz. The code was compiled with the IBM C compiler version 3.6.6 and linked with TreadMarks runtime library verion 1.0.3.3-BETA. The following compiler options were used : `-O3 -qstrict -qarch=pwr2 -qtune=pwr2`

TreadMarks is a software distributed shared memory system and it has been deployed on a machine with a slower interconnect for this work; therefore it is not a surprise that the speedups obtained are generally low, and in particular, much lower than those on the SGI Origin. But the gains in speedup resulting from a distribution and privatization of data are

```
for(ck=0,k=Tmk_proc_id; ck < N-1 ;ck++)
{
  if(ck == k)
  {
    for(j=ck+1;j<N;j++)
      shared->lu[j] = lu[ck/Tmk_nprocs][j] =
         lu[ck/Tmk_nprocs][j] / lu[ck/Tmk_nprocs][ck];

     k = k + Tmk_nprocs;
   }
   Tmk_barrier(1);
   /* update right part. */
   for(j=k;j<N;j+=Tmk_nprocs)
     for(i=ck+1;i<N;i++)
       lu[j/Tmk_nprocs][i] = lu[j/Tmk_nprocs][i] -
                               lu[j/Tmk_nprocs][ck]*shared->lu[i];
   Tmk_barrier(2);
}
```

Program 13. LU Factorization with TreadMarks

greater in this case, since there is a much higher difference in cost between local and remote memory access. A comparison of the results demonstrates effectively that this programming style can make a significant difference, even for software distributed shared-memory systems.

For the Jacobi kernel, the privatized version outperforms the shared version by more than a factor of 5 on 16 threads; for the LU code, it is a factor of 4. The difference is not so large for the LBE kernel, and corresponds to the results on the SGI Origin.

5 Role of Language Extensions

Few applications are as easy to parallelize as the Jacobi stencil, where there is little sharing of data, and a single thread may perform all assignments to an array element for the entire duration of the program, without leading to any load balancing problems. Yet even in this case, the adoption of the alternative "SPMD" programming style led consistently to significantly better performance. In particular, we achieved greater scalability (for the same data set) on all three codes. Most applications are likely to have either load balancing problems, as apparent in the LU kernel, or changes in the relationship between data and threads, as in the LBE application.

The difficulty with the SPMD programming style adopted in the second set of experiments is that it requires the user to consider the needs of an entire application, in order to decompose data and privatize it. But one of the benefits of OpenMP is thereby lost. Although we cannot entirely eliminate this problem, since we expect the user to supply appropriate data distributions, we do believe that it should be possible to provide support for the translation of a program from the initial loop-level parallel version to an equivalent SPMD-style program. However, the translation process relies on user directives that specify the distribution of data and its relationship to the computation.

The language features provided by vendors include the most important of these directives, namely those to distribute and redistribute data, as well as those that bind loop iterations to a thread based upon the distribution of the data it updates. In addition to these, we expect that a general block distribution, such as that available in HPF-2, would be a useful addition to the set of distribution kinds. It can be used to create a balanced distribution for unstructured meshes, so long as these are renumbered.

Our set of data features, described in a paper currently under preparation, includes syntax for each of these, where the form borrows heavily from HPF. It also includes a `SHADOW` directive, with which the user may give the compiler information on the amount of data that needs to be exchanged with "neighboring" threads. In contrast to the manner in which HPF handles mappings at subroutine boundaries, it does not distinguish between prescriptive, descriptive and transcriptive distributions.

It is interesting to note that SGI maps data to processors, whereas Compaq refers to node memories. We prefer the former for two reasons: first, unlike the latter, it does not assume that nodes have the same number of processors throughout the system – even monolithic machines are increasingly heterogeneous. Second, where elementwise distributions are concerned, there will be additional reduction in false sharing if each processor has its own pages of data. A disadvantage of this approach, however, is that this does not enable us to specify (indirectly) that a set of threads should execute on the same node of a larger system, or more generally, that they should be placed near each other. Other researchers have attempted to address this issue separately, for example by considering the grouping of threads [2].

Finally, it is vital that we do not introduce OpenMP extensions in such a way that incremental program development is no longer possible. Although performance problems may arise as the result of a partially parallelized program, it is essential that it executes correctly and that the parallelization directives are followed where inserted, so that their performance implications for that region of code may be studied. This requires careful consideration of the rules for their translation, especially at subroutine boundaries.

6 Related Work

A number of reports on OpenMP application development are now available [8] and there have been several comparisons of OpenMP with other programming models on ccNUMA architectures [26]. Some include experiences using a mixed programming style for clusters of SMPs, MPI and OpenMP, to parallelize applications [28, 9, 14]. As a result of these experiences, various studies have also considered extensions of OpenMP for NUMA machines or clusters. Whether or not they are needed, is a matter of current debate.

Omni OpenMP [23, 22] is an OpenMP compiler that translates OpenMP to distributed memory systems, possibly with SMP nodes, via its own software distributed shared memory system. It does not include any directives for the mapping of data. We found that directives improved the performance of applications under TreadMarks, a system similar in spirit to Omni.

In [2], the authors present some extensions to OpenMP that enable the definition of groups of threads and the assignment of work to these groups. This allows the user to control the allocation of the work in a hierarchical manner.

In [20], the authors note the different memory latencies on Origin platforms, but argue that their experiments show there is no need for data distribution in OpenMP. Their work emphasizes the importance of using the right distribution from the beginning (via a first-touch policy), and we do not disagree on this aspect. However, the experiments do not go beyond 16 processors, so the problems introduced by the increased number of routers on larger configurations are likely to have little impact. A fundamental weakness of their argument, however, is that they do not compare their results with those that can be obtained by using the SGI extensions, or with any other kind of mapping extensions.

As we have already discussed in this article, both SGI and Compaq have defined extensions to OpenMP for their respective systems. We believe that there is scope for improving the set of directives provided by SGI; in particular, we think that the `SHADOW` directive could be crucial for obtaining higher performance. Compaq has introduced a much larger set of extensions in their OpenMP compiler [5, 6], although the core features are very similar. Although their features provide greater expressivity, we feel that it is too rich to enable easy porting to other architectures.

Finally, Portland Group Inc. (PGI) proposes extensions to OpenMP for clusters of SMP systems in a language named Distributed OpenMP (DoMP) [16, 17]. They are also considered suitable for ccNUMA and distributed memory architectures. Their set includes the `DISTRIBUTE` and `ON HOME` directives, where distributions are to node rather than processor memory, in order to reflect the hierarchy of node clusters.
In contrast to the other proposals, they adopt an HPF execution model outside `PARALLEL` regions, and a distributed OpenMP execution model inside a `PARALLEL` region. Under this model, a distributed OpenMP thread is in fact constituted of multiple threads, one on each node. One-sided communication between them is needed to ensure access to remote memories.

Each of these approaches includes interesting functionality. However, there is not yet agreement on whether or not these features are needed, let alone which features are really needed to provide a good OpenMP translation for ccNUMA and clustered systems, whether or not this occurs in the form of a source-to-source translation or direct compilation. Where there is similarity, it is obscured by differences in syntax. Given that OpenMP was designed to be portable, it is desirable that a minimal set of extensions emerge from these which is both useful and widely applicable.

7 Conclusions and Future Work

OpenMP [21] is a set of directives for developing shared memory parallel programs. As such, it does not includes features for data distribution, since they are not needed when memory access is uniform across the target system. It is an effective programming model for developing codes that are to run on small shared memory systems. It is also a promising alternative to MPI for codes running on ccNUMA platforms – or it would be if it could provide similar levels of performance on these and true distributed memory architectures.

We have shown the performance benefits of a programming style that privatizes data in the experiments discussed in this paper. This style relies on the partitioning of global data to create local, private data for each thread, and the corresponding adaptation of loop nests. It also requires the explicit construction of buffer arrays for the transfer of data between threads. It further obtains performance by mapping loop iterations to threads for which specified data is local, in the sense that it is stored in local memory. Each of these actions resembles part of the HPF translation process. HPF provides directives that can be used to direct a compiler or tool within the context of OpenMP also. The most important of these are the (`DISTRIBUTE`), (`ON HOME`), and `SHADOW` directives. At the University of Houston, we are beginning to develop a source-to-source compiler for OpenMP that will accept such directives, in order to further test our ideas. Although we plan to target ccNUMA architectures, we hope that subsequent efforts will focus on cluster based architectures.

Acknowledgements

The authors wish to thank Li-Shi Luo for the provision of the LBE code that was by far the most interesting of those studied in this paper, and to thank Li-Shi, Seth Milder and Piyush Mehrotra, all at ICASE, NASA Langley Research Center, for their help in understanding the application and the problems it poses. They are also grateful to Jerry Yan and Michael Frumkin for their encouragement to investigate performance problems related to this architecture, and to Tor Sorevik, who introduced the first author to OpenMP and its intricacies on the Origin 2000.

References

[1] C. Amza, A. Cox, and et al. Treadmarks: Shared memory computing on networks of workstations. IEEE Computer, 29(2):18–28, February 1996.

[2] E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro, and J. Oliver. NanosCompiler. A Research Platform for OpenMP extensions. In *First European Workshop on OpenMP - EWOMP'99*, Lund University, Lund, Sweden, 1999.

[3] R. Baxter, P. Graham and M. Bowers. Rapid Parallelisation of the Industrial Modelling Code PZFlex. In *EWOMP 2000, European Workshop on OpenMP*, Edimburgh, Scotland, U.K., September 2000.

[4] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory using multiprotocol release consistency, 1991.

[5] J. Bircsak, P. Craig, R. Crowel, J. Harris, C.A. Nelson, and C.D. Offner. Extending OpenMP For NUMA Machines: The Language. In *WOMPAT 2000, Workshop on OpenMP Applications and Tools*, San Diego Supercomputer Center, San Diego, California, July 2000.

[6] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, and C.D. Offner. Extending OpenMP for NUMA Machines. In *SC2000, Supercomputing*, Dallas, Texas, USA, November 2000.

[7] R. Blikberg and T. Sorevik. Early experiences with OpenMP on the Origin 2000. Proc. European Cray MPP meeting, Munich, Sept 1998

[8] M. Brorsson and B. Chapman (Eds.) Selected Papers from the First European Workshop on OpenMP. Special Issue of *Concurrency: Practice and Experience* 12, 2000

[9] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *SC2000, Supercomputing*, Dallas, Texas, USA, November 2000.

[10] T. Faulkner. Performance Implications of Process and Memory Placement using a Multi-Level Parallel Programming Model on the Cray Origin 2000. Available at http://www.nas.nasa.gov/~faulkner

[11] HPF Forum. High Performance Fortran language specification, Version 2.0, January 1997.

[12] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *19th Annual International Symposium on Computer Architecture*, pages 12–21, May 1992.

[13] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Winter Usenix Conference*, pages 115–131, January 1994.

[14] P. Kloos and F. Mathey and P. Blaise. OpenMP and MPI programming with a CG algorithm. In *EWOMP 2000, European Workshop on OpenMP*, Edimburgh, Scotland, U.K., September 2000.

[15] J. Laudon and D. Lenoski. The SGI Origin ccNUMA Highly Scalable Server. SGI Publishe White Paper, March 1997.

[16] M. Leair, J. Merlin, S. Nakamoto, V. Schuster, and M. Wolfe. Distributed OMP - A Programming Model For SMP Clusters. In *CPC2000, Compilers for Parallel Computers*, Aussois, France, January 2000.

[17] J. Merlin (The Portland Group, Inc.). Distributed OpenMP: Extensions to OpenMP for SMP Clusters. In *EWOMP 2000, European Workshop on OpenMP*, Edimburgh, Scotland, U.K., September 2000.

[18] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Version 1.1, June 1995.

[19] B. Nichols, D. Buttlar, and J. Proulx Farrell. Pthreads Programming. O'Reilly publishers, 1996

[20] D.S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is Data Distribution Necessary in OpenMP? In *SC2000, Supercomputing*, Dallas, Texas, USA, November 2000.

[21] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 2.0, November 2000.

[22] M. Sato, H. Harada, and Y. Ishikawa. OpenMP compiler for a Software Distributed Shared Memory System SCASH. In *WOMPAT 2000*, San Diego, July 2000.

[23] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *EWOMP'99, European Workshop on OpenMP*, pages 32–39, Lund, Sweden, September 1999.

[24] A. Sawdey. SC-MICOM. Software and documentation available from `ftp://ftp-mount.ee.umn.edu/pub/ocean/`

[25] S. Seidel. Access Delays Related to the Main Memory Hierarchy on the SGI Origin2000. In *Third European CRAY-SGI Workshop*, Paris, France, September 1997.

[26] H. Shan and J. Singh. A Comparison of MPI, SHMEM and Cache-Coherent Shared Address Space Programming Models on the SGI Origin 2000. Proc. Int. Conf. on Supercomputing, 1999

[27] Silicon Graphics Inc. MIPSPro Fortran 90 Commands and Directives Reference Manual. Document number 007-3696-003. Search keyword MIPSPro Fortran 90 on http://techpubs.sgi.com/library/.

[28] L.A. Smith and J.M. Bull. Development of Mixed Mode MPI/OpenMP Applications. In *WOMPAT 2000*, San Diego, July 2000.

[29] A.J. Wallcraft. SPMD OpenMP vs MPI for Ocean Models. In *First European Workshop on OpenMP - EWOMP'99*, Lund University, Lund, Sweden, 1999.

[30] X3H5 Committee. Parallel Extensions for Fortran. TR X3H5/93-SD1 Revision M, ANSI Accredited Standards Committee X3, April 1994

[31] X. He and L.-S. Luo. Theory of the Lattice Boltzmann Method: From the Boltzmann Equation to the Lattice Boltzmann Equation. Phys. Rev. Lett. E, 56(6), 6811, 1997

[32] W. Zwaenepoel et al. Concurrent Programming with TreadMarks. TreadMarks users Manual. http://www.cs.rice.edu/ willy/papers/doc.ps.gz.