Performance Oriented Programming for NUMA Architechtures *

B. Chapman, A. Patil and A. Prabhakar

Department of Computer Science, University of Houston, Houston, TX
{chapman,amit,achal}@cs.uh.edu

Abstract. OpenMP is emerging as a viable high-level programming model for shared memory parallel systems. Although it has also been implemented on ccNUMA architectures, it is hard to obtain high performance on such systems, particularly when large numbers of threads are involved. Moreover, it is applicable to NUMA machines only if a software DSM system is present. In this paper, we discuss various ways in which OpenMP may be used on ccNUMA and NUMA architectures, and evaluate several programming styles on the SGI Origin 2000, and on TreadMarks, a Software Distributed Shared Memory System from Rice University. These results have encouraged us to begin work on a compiler that accepts an extended OpenMP and translates such code to an equivalent version that provides superior performance on both of these platforms.

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

1 Introduction

Among the various programming models available for parallel programming, the shared memory programming model provides ease of programming, as the programmer is freed from the intricacies of communication and data collection. It is also much easier for a compiler to generate parallel code for a shared memory machine from a sequential program with programmer directives than for a distributed memory platform.
However, pure SMPs do not scale beyond 8 or 16 processors unless a prohibitively expensive crossbar is deployed. When more processors are added, the shared memory bus architecture of the overwhelming majority of commercial systems

* 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 the authors were in residence at ICASE, NASA Langley Research Center. These sources of support are gratefully acknowledged.

presents a serious bandwidth bottleneck. In response, vendors such as SGI, Compaq, Sun and HP have built systems partitioned into smaller modules. Each such module is a pure SMP with a high speed interconnect linking it to all other modules, either directly or indirectly. The resulting system, potentially comprised of a large number of modules, is characterized by a memory hierarchy with non-uniform memory access times (NUMA). Systems which provide cache coherency are called cc-NUMA. Commercial examples include: SGI's Origin 2000, Compaq's AlphaServer GS80, GS106 and GS320. On such platforms, processes may directly access data in memory across the entire machine via load and store operations.
On the other hand, clusters of uni-processor or multi-processor machines have physically distinct memories with no hardware support for coherency. It is possible to consider such clusters to be shared memory machines, if we implement a layer of software on them that manages memory consistency across the physically distributed memories [1, 12, 2, 13]. The characteristics of distributed memory systems with a Software Distributed Shared Memory (SDSM) layer on top are similar to cc-NUMA machines.

In our work, we consider how to write scalable OpenMP applications despite the non-uniformity of memory accesses. We show a programming style in which the locality of data and work is taken into account. Unfortunately, this programming style involves making many changes to a program's code, and thus it reduces the benefits of OpenMP with respect to ease of programming. We believe that it is possible to translate a standard OpenMP program with a few extensions into a higher-performing equivalent code for ccNUMA platforms. The paper outlines the language extensions needed to support this task. The paper is organized as follows: we first describe ccNUMA architectures and then discuss OpenMP language extensions and strategies provided by the vendors to support efficient execution on ccNUMA systems. We then introduce our experiments and show the superiority of the coding style that we plan to generate from a simple set of such language extensions.

2 ccNUMA Architectures

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.

Our experiments have been performed on the Silicon Graphics' Origin 2000, a representative of such systems [9]. It is organized as a hypercube, where each node typically consists of 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 connects 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 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. It attempts to allocate memory for a process on the same node on which it runs. However, results are not guaranteed. 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.

2.1 TreadMarks: Software Distributed Shared Memory System

TreadMarks [1, 12] 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) [3] called the Lazy Release Consistency protocol [8].

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) [8] 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 behavior of a distributed memory platform programmed via the TreadMarks system is similar in spirit to that of 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.

3 OpenMP

OpenMP consists of a set of compiler directives, as well as library routines, for explicit shared memory parallel programming. The directives and routines 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. It provides a familiar programming model, enables relatively fast, incremental and portable application development, and has thus rapidly gained acceptance by users. The OpenMP directives may be used to declare parallel regions, to specify the sharing of work among threads, and for synchronizing threads. 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.

3.1 OpenMP Language Extensions for ccNUMA Platforms

Although OpenMP can be transparently implemented on a ccNUMA platform, as well as mapped to a SDSM system such as TreadMarks, it does not account for non-uniformity of memory access by design. Therefore, the user cannot explicitly specify that data should be allocated on or near a node where computations based upon it are performed; nor can the user explicitly prefetch data during execution within OpenMP.

The best solution to the problem of co-allocating data and threads in an OpenMP program would be to implement a transparent, and highly optimized, dynamic migration of data. However, it is very hard for the operating system to determine when to migrate data and current commercial implementations do not perform particularly well. Moreover, page-based storage is not always a suitable basis for an appropriate distribution of data. Both SGI and Compaq thus provide low-level features for directly influencing the location of pages in memory, as well as high level directives to specify data distribution and thread scheduling in OpenMP programs [4, 7]. 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 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 some limitations on the use of elementwise distributed arrays [7], 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`.

We have begun to implement a set of directives that is similar to the above; indeed, we have used them as the starting point for our work. We aim to use directives that are as simple as possible without sacrificing important functionality. Our set therefore includes the above data distribution features, but also includes a general block distribution that may help map the data of some unstructured codes.

The use of a `PROCESSORS` directive enables load-balanced mappings to be specified for systems where the nodes are heterogeneous, but it does not capture the structure of a hierarchical system. In contrast, the `MEMORIES` directive enables one to identify and target the individual SMPs in a large system, but causes problems if the individual nodes have different numbers of configured processors. A general block distribution can help achieve load balance in the latter case, so would seem to provide a way out of this apparent dilemma. However, our compilation strategy considers data to be ultimately mapped to threads, since it can then be privatized.

An `ON HOME` directive similar to that of Compaq maps iterations or parallel sections to a processor associated with the memory a particular datum is stored on, and thus augments OpenMP's own mechanisms for distributing the computation among threads. If no mapping is explicitly specified, OpenMP defaults

prevail. Further, our set of directives includes the `SHADOW` directive borrowed from HPF. This directive enables the user to specify the extent of non-local data accessed by a thread, whether it is read or written, during the computation assigned to it. It will be used by the compiler to set up buffers for storing and copying this data.

Our goal is to translate these directives to an SPMD-style of program. The approach was motivated not only by a variety of previous reports on OpenMP programming experiences, but also by experiments we have conducted using OpenMP on an Origin and on top of TreadMarks. We describe these experiments below. Although the set of programs is small, the examples were chosen to reflect different kinds of data access patterns. On-going work investigates the translation of more substantial codes. Implementation of an initial set of directives has begun in the Cougar compiler under development at the University of Houston.

4 Standard OpenMP Programming

We describe experiments using three sample applications written in OpenMP in a straightforward loop-level parallel programming style. For our ccNUMA platform, we relied on the Origin 2000 systems at the National Center for Supercomputing Applications (NCSA). They were compiled with SGI's MIPSpro7 Fortran 90 compiler under the options `-mp -64 -mips4 -r1000 -Ofast -IPA` and run in multiuser mode. We made use of the various strategies provided by SGI to obtain good performance under OpenMP. Initially, the most suitable "first touch" default strategies, whereby a datum is stored on the node where it is first accessed, were used in each case. In all but one case, the results shown were obtained with the Origin's `_DSM_MIGRATION` (page migration) environment variable was set to OFF. One set of results with this switch enabled is provided, so that the reader may compare them.

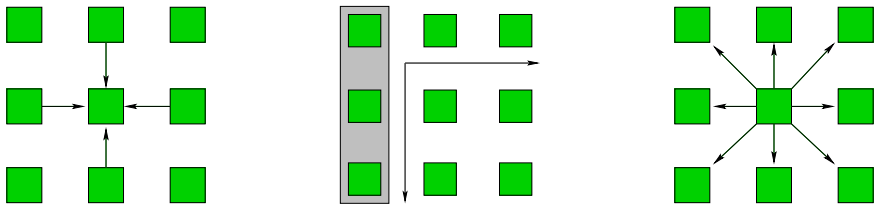
The SDSM experiments make use of the SDSM system TreadMarks, which was installed on an IBM SP2 system at the University of Houston; it operated on thin nodes with 128MB memory and running at 120MHz. The code was compiled with the IBM C compiler version 3.6.6 and linked with TreadMarks runtime library verion 1.0.3.3-BETA. The compiler options used were `-O3 -qstrict -qarch=pwr2 -qtune=pwr2`.
This platform has a relatively slow interconnect; 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 that we shall see as a result of a distribution and privatization of data are greater in this case, since there is a much higher difference in cost between local and remote memory access.

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 or multiprocessing code option on the O2000). We have not included results for more than 64 (Jacobi) and

32 (LU, LBE) processors respectively, since for the matrix sizes used, there was not enough computation remaining to keep additional threads busy.

4.1 Jacobi

The Jacobi method is one of the simplest numerical solvers for partial differential equations. Although it converges very slowly, it exhibits excellent spatial locality. In this short code, almost all data accesses can be made local (see Prog. 1 and Fig. 1 (a)). Thus it should be possible for a system to obtain good performance on a Jacobi stencil under OpenMP without user-specified distribution directives. On the Origin, we may exploit the 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. We give timings for the default allocation as well as for columnwise BLOCK distribution of matrices A and B using SGI's `DISTRIBUTE` and `DISTRIBUTE_RESHAPE` directives.



(a) **(b)** **(c)**

Figure 1. Jacobi (a), LU (b) and LBE (c) access stencils

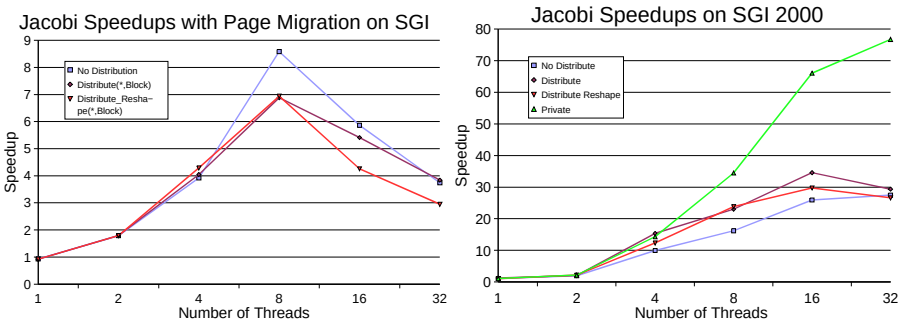
```
!$OMP PARALLEL DO
  do j = 1,n
     do i = 1,n
        A(i,j) = (B(i-1,j)+B(i+1,j)+B(i,j-1)+B(i,j+1)) * c
     end do
  end do
!$OMP END PARALLEL DO
```

Program 1. Standard OpenMP version of Jacobi Kernel

The TreadMarks version of the Jacobi kernel is a straightforward translation of the corresponding OpenMP version. Shared variables are allocated explicitly using the `Tmk_malloc` primitive. Iteration space is block divided and synchronization is achieved through calls to the `Tmk_barrier` function.

The performance figures shown are based on runs with a 1024 by 1024 matrix of double precision data. For this data size, the local portion of the matrix fits precisely into a number of pages for all the numbers of threads used. Note, however, that only the elementwise `DISTRIBUTE_RESHAPE` directive guarantees that data will start on a page boundary. No matrix element is updated by more than one thread. 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.

We first show results obtained with the `_DSM_MIGRATION` switch set on (Fig. 2 (a)). If we compare this with our other results (Fig. 2 (b)), it is clear that pages are indeed being migrated; they will have to migrate back to their previous location for updating. With an increasing number of processors, more and more pages are moved; from 16 processors onward, some data will move across one or more routers.



(a) **(b)**

Figure 2. OpenMP Jacobi speedups on SGI O2000 with page migration (a) without page migration (b)

The first touch policy (Prog. 1) and `DISTRIBUTE (*,BLOCK)` 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 on the Origin.

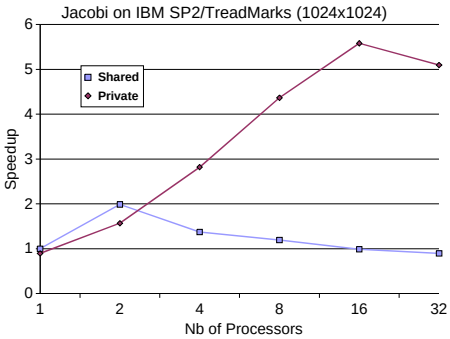


Figure 3. Jacobi speedups on IBM SP2 with TreadMarks

The speedups obtained with Treadmarks, shown as "shared" code in Fig. 3 are not impressive. In fact, there is a slowdown for more than 16 processors. This is attributed mainly to the high coherency maintenance overheads. Many pages are shared and TreadMarks has to collect and disseminate coherency data for all of them at each synchronization point. We discuss the "private" version further below.

4.2 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. 2).

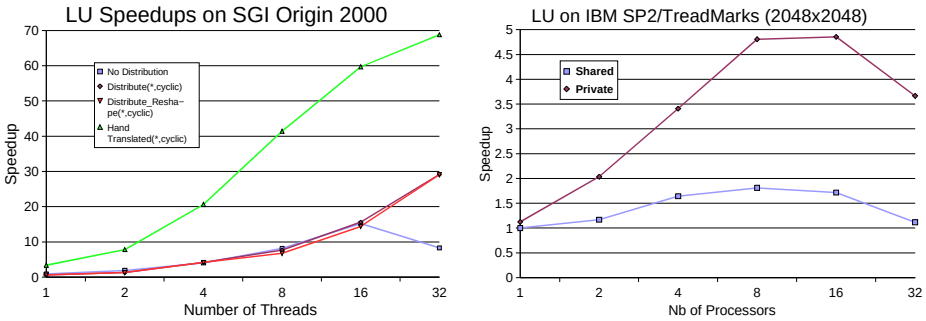
The matrix size operated on reduces progressively (see Fig. 1 (b)). Therefore, we ensure that all threads continue to participate in the computation and ensure good load balancing by using a `CYCLIC` distribution. This can be achieved using the page-based `DISTRIBUTE` directive or the elementwise `DISTRIBUTE_RESHAPE` 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 would assign iterations to threads by block. Timings for the code without the `AFFINITY` directive were approximately 40 % higher than those shown here.

In the TreadMarks version, compute matrix lu is declared shared and iterations are block distributed. The normalization code is protected by `Tmk_lock_acquire` and `Tmk_lock_release` primitives.

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 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

```
!$OMP PARALLEL
  do k = 1,n-1
!$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 2. Standard OpenMP versions of LU



(a) (b)

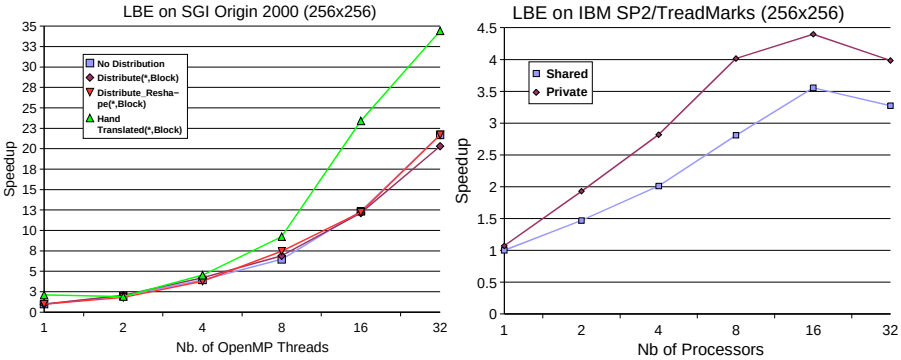
Figure 4. LU speedups (a) on SGI O2000 (b) IBM SP2 with TreadMarks

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.
The inherent load imbalance in the LU algorithm leads to contention at the node that performs the normalization. Coupled with the high coherence overheads due to the large number of shared pages, this lead to the performance degradation in the TreadMarks version.

4.3 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 [16]. 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. 1 (c)). The `Collision_advection_interior` subroutine with a write shared data access is shown in Program 3. 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. As before, we developed three versions

of this code. 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 TreadMarks version for the LBE solver is similar in spirit to the Jacobi code.



(a) **(b)**

Figure 5. LBE speedups (a) on SGI O2000 (b) on IBM SP2 with TreadMarks

```
!$OMP PARALLEL
  do iter = 1, niters
     Calculate_ux_uy_p
     Collision_advection_interior
     Collision_advection_boundary
  end do
!$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 3. OpenMP version of the LBE Algorithm and Collision_advection_interior Kernel

The experiments were performed with a 256 by 256 matrix (see Fig. 5). On the Origin, all three versions behaved similarly up to 32 processors. 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 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.
The speedups obtained for the TreadMarks version of LBE are significantly better than Jacobi and LU. They increase consistently till 16 processors, beyond which the per processor computation is not enough to offset the communication overhead.

5 An Alternative OpenMP Programming Style for ccNUMA Platforms

Under the SPMD parallelization strategy, we distribute the arrays 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 [15].

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. [10, 14, 6, 11]). In [15], 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.

In the corresponding TreadMarks code versions, labeled *private* in the performance figures shown above, the arrays are privatized to realize this SPMD style (cf. the hand-coded versions presented below). Codes are y similar to the corresponding Origin versions (e.g. LU code in Prog. 6for TreadMarks and in Prog. 5 for SPMD OpenMP).The results we obtained are given in Figures 3 for Jacobi, 4 (b) for LU and 5 (b) for LBE.

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). 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. At the start of the PARALLEL region the 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 executing the code to calculate its own portion of data.

```
!$OMP PARALLEL SHARED(buflower,bufupper,chunk) &
!$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 4. Jacobi Kernel in SPMD style

This version of the program shows superlinear speedup (Fig. 2 (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 result of the decreasing computation per thread.
The TreadMarks results for the private version show speedups which are much better than the shared version. Increased data locality and lower coherency overheads improve the computaion to communication ratio i.e more percentage of the execution time is spent doing useful computation. For the private case, we get an almost linear speedup upto 16 processors. With 32 processors there is a decrease in speedup.

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. 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.

In the PARALLEL region (see Prog. 5), 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.

```
!$OMP PARALLEL SHARED(shbuf)&
!$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 5. LU Kernel in SPMD style

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. 4 (a)) can be explained by the cache effect and efficient handling of the shared buffer updates.

For the TreadMarks version only one column was declared shared, the rest of the array was divided cyclicly and made totally private. The iteration space was also divided in cyclic fashion. This version is consistently better than the shared version.

```
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 6. LU Factorization with TreadMarks

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.

The SPMD version of LBE shows a remarkable increase in performance for 16 and 32 processors (Fig. 5 (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 required the selection of a distribution strategy for the program's data. After computing the local size of data objects, including shadow regions, it is a matter of introducing buffer copying and synchronization to ensure that data is read from buffers after it has been written to them.

A comparison of the different code versions under TreadMarks demonstrates 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.

6 Conclusions and Future Work

OpenMP [5] is a set of directives for developing shared memory parallel programs. 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 together with ease of programming.

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. At the University of Houston, we are beginning to develop a source-to-source compiler for OpenMP that will accept a modest set of extensions to OpenMP and use them to generate an OpenMP code written in this style. The most important of these are the (`DISTRIBUTE`), (`ON HOME`), and `SHADOW` directives.

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 Piyush Mehrotra and Seth Milder 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. J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors, 1989.
3. J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory using multiprotocol release consistency, 1991.

4. 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.
5. OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 2.0, November 2000.
6. F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *SC2000, Supercomputing*, Dallas, Texas, USA, November 2000.
7. 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/.
8. 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.
9. J. Laudon and D. Lenoski. The SGI Origin ccNUMA Highly Scalable Server. SGI Publishe White Paper, March 1997.
10. 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.
11. R. Blikberg and T. Sorevik. Early experiences with OpenMP on the Origin 2000. In *Proc. European Cray MPP meeting*, Munich, September 1998.
12. Concurrent Programming with TreadMarks. TreadMarks users Manual. http://www.cs.rice.edu/ willy/papers/doc.ps.gz.
13. M. Sato, H. Harada, and Y. Ishikawa. OpenMP compiler for a Software Distributed Shared Memory System SCASH. In *WOMPAT 2000*, San Diego, July 2000.
14. L.A. Smith and J.M. Bull. Development of Mixed Mode MPI/OpenMP Applications. In *WOMPAT 2000*, San Diego, July 2000.
15. A.J. Wallcraft. SPMD OpenMP vs MPI for Ocean Models. In *First European Workshop on OpenMP - EWOMP'99*, Lund University, Lund, Sweden, 1999.
16. X. He and L.-S. Luo. Theory of the Lattice Boltzmann Method: From the Boltzmann Equation to the Lattice Boltzmann Equation. In *Phys. Rev. Lett. E, 56(6), 6811*, 1997.