Program Development Environment for OpenMP Programs on ccNUMA Architechtures *

B. Chapman, O. Hernandez, A. Patil and A. Prabhakar
Department of Computer Science, University of Houston, Houston, TX
{chapman,oscar,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. In this paper, we discuss ways in which OpenMP may be used on ccNUMA architectures, and describe a programming style that can provide scalable high performance on such systems. We give an example of its use on the SGI Origin 2000, and on TreadMarks, a software DSM system from Rice University. These results have encouraged us to work on a programming environment that provides general support for OpenMP application development and incorporates a system to translate standard loop-level parallel OpenMP code, with additional user input in the form of directives, into an equivalent OpenMP program relying on an alternative programming style. The equivalent program does not use constructs external to OpenMP.

Keywords: shared memory parallelism, parallel programming, OpenMP, ccNUMA architectures, restructuring, data locality, data distribution, software distributed shared memory, programming environments

1 Introduction

The shared memory programming model provides ease of programming, and is the model of choice for many developers of parallel code. However, bus-based SMPs do not scale beyond 8 or 16 processors. In response, vendors such as SGI, Compaq, Sun and HP build machines consisting of SMP modules linked by a high speed interconnect. The result is a non-uniform memory access (NUMA) system; those providing 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. Clusters of uni- or multiprocessor machines with no hardware support for coherency may also be programmed as shared memory machines, if a layer of software is provided on them to manage consistency across the physically distributed memories [1, 11]. Such platforms with a software Distributed Shared Memory (SDSM) layer are similar to cc-NUMA machines.

*This work was partially supported by the DOE under the Los Alamos Computer Science Institute and by NSF under grant number NSF ACI 99-82160. These sources of support are gratefully acknowledged.

We are working to provide support for the creation of OpenMP programs for shared memory and ccNUMA platforms. To this end, we are developing a tool that enables users to understand the salient aspects of a program, as well as providing explicit help for migration to OpenMP. The effort reported upon in this paper was motivated by our desire to help application developers write scalable OpenMP applications for ccNUMA machines despite the non-uniformity of memory accesses. We show that OpenMP can be used in a fashion that takes the locality of data and work into account. Unfortunately, this involves making many changes to a program's code. We attempt to minimize the effort required by automatically translating OpenMP programs into a suitable form. The tool requires additional user input in order to do so.

The paper is organized as follows: we first introduce the OpenMP programming language in Section 2 and then describe ccNUMA architectures and its execution on such systems (Section 3). An example is given of an application and its performance under two distinct OpenMP programming styles in Section 4. We then describe our tool, the Cougar compiler. The paper concludes with a brief discussion of related work and some conclusions.

2 OpenMP

OpenMP consists of a set of compiler directives and 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 based upon fork-join parallelism, 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. Features for coordinating threads include barriers, locks, atomic, and single-threaded execution.

Data may be shared between threads, or may be private to a thread. Since there is a performance penalty to be paid whenever data needed by a thread resides in another thread's cache, such interference should be minimized. Mechanisms for influencing this include indirect means such as padding array sizes to separate memory references onto distinct pages, as well as explicit privatization. Work outside parallel regions is executed by the master thread only; thus it is important that as much of the computation as possible is within worksharing constructs contained in parallel regions.

3 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 machine's globally addressable memory. It is organized hierarchically, with 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 [8]. 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 via a switch-based interconnect. Each processor has two levels of cache. 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 2 to nearly 4 times the cost of access to local memory. The experienced cost of a remote memory access depends on contention for bandwidth also. 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.

3.1 TreadMarks: Software Distributed Shared Memory System

TreadMarks [1] is a Software Distributed Shared Memory system (SDSM) that uses the operating system's virtual memory interface to implement the shared memory abstraction. It employs an extension to the Release Consistency protocol (RC) called the Lazy Release Consistency protocol [7]. LRC aims to reduce the number of messages and amount of data transferred by postponing the propagation of modifications until a page is acquired. The acquiring processor must determine which modifications it needs from which processors. TreadMarks uses the multiple-writer protocol, so two or more processors may simultaneously update data on the same page. When the processors reach a synchronization point, they exchange their modifications. The behavior of a distributed memory platform programmed via the TreadMarks system is similar in spirit to that of ccNUMA systems.

3.1.1 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. Therefore, the user cannot explicitly specify that data should be co-located with a thread that updates it. 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 [3, 6]. The extension sets differ, although there is substantial overlap in the core functionality. A major component of both is the `DISTRIBUTE` directive. This specifies the manner in which a data object is mapped onto the system. Each dimension of an array may be distributed in `BLOCK`, `CYCLIC` or `*` fashion. 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,*)
```

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. Their advantage is that they can be added to an existing program without any restrictions, since they do not change the layout of the data object. A major disadvantage is that they are unsuitable for distributing small arrays.

It is also possible to perform data distribution at element granularity as in HPF, rather than page granularity. This involves rearranging the array's layout in memory so that two elements which should be placed in different memories are stored in separate pages. The resulting layout guarantees the specified distribution at the element level, but it may violate the language standard. There are some limitations on the use of elementwise distributed arrays [6] as a result.

Both vendors also supply directives to associate computations with the location of data in storage. Compaq's `ON HOME` directive informs the compiler exactly how to distribute iterations of parallel loops over memories. SGI similarly provides `AFFINITY`, a directive that can be used to specify the distribution of loop iterations based on either `DATA` or `THREAD` affinity.

4 An Experiment with OpenMP Programming Styles

We have performed experiments using example OpenMP applications written in a straightforward loop-level parallel programming style and in an SPMD programming style on the Origin 2000 at NCSA, and on an IBM SP2 with TreadMarks. We show results for one of the codes below. Speedup figures have been normalized. They are given with respect to the serial time of the initial OpenMP version (without vendor directives or multiprocessing option).

4.1 LBE

LBE is a computational fluid dynamics code that solves the Lattice Boltzmann equation. It was provided by Li-Shi Luo of ICASE, NASA Langley Research Center [14]. The numerical solver employed by this code uses a 9-point stencil. Its execution is complicated by the fact that an iteration updating data at one point will also update other points in this stencil. As a result different loop iterations, possibly executed by distinct threads, may update the same point. The `Collision_advection_interior` subroutine with a shared write access is shown in Program 1. The Origin's 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 first developed three versions of this code relying on techniques provided by the vendor. Results are shown for a 256 by 256 matrix in Fig. 1. The first of these relies on SGI's default first touch policy to allocate data according to its initial usage. Although this should result in good locality, cache lines containing data updated by multiple threads will be migrated between them periodically. This code version is labeled "No Distribution". The second version uses SGI's `DISTRIBUTE (*,*,BLOCK)` directive to distribute pages of the matrices by block in the last dimension. This will result in exactly the same distribution of data. Although the `DISTRIBUTE_RESHAPE` directive, used with the same distribution in the third version, will provide a more precise mapping, it cannot alleviate the problem of data sharing between the threads. All three versions behaved similarly up to 32 processors, so the directives have not improved performance of the code.

The "shared" TreadMarks version of the LBE code is a straightforward translation of the initial OpenMP code to the TreadMarks API. Shared variables are allocated explicitly using the `Tmk_malloc` primitive. The iteration space is block divided and synchronization is achieved through calls to the `Tmk_barrier` function. Performance on this unoptimized implementation of TreadMarks is significantly lower than on the Origin, although it does remain consistent up to 16 processors, beyond which the per processor computation is not enough to offset the communication overhead.

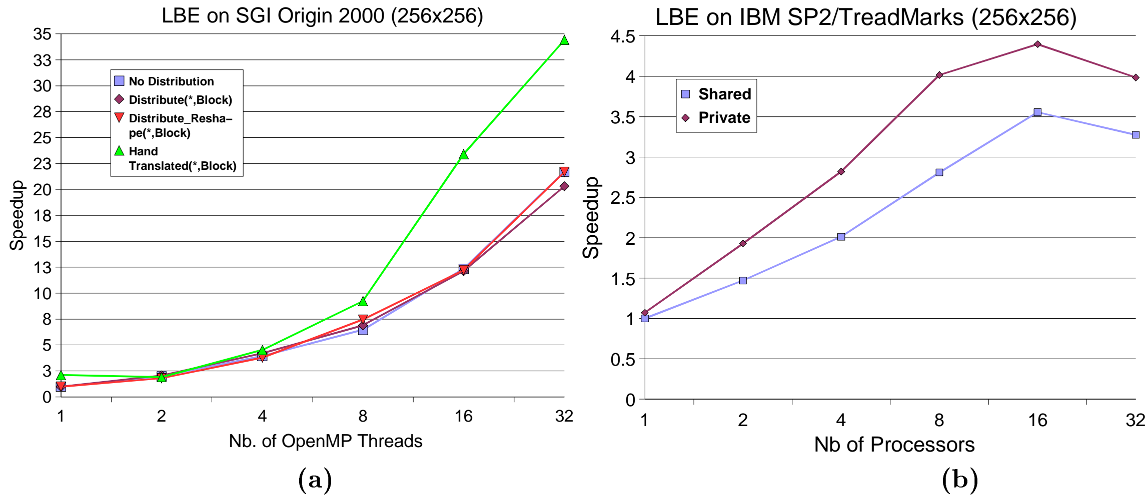

Figure 1. LBE speedups (a) on SGI O2000 (b) on IBM SP2 with TreadMarks

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

The "private" version of LBE on TreadMarks, and the hand translated version on the Origin, are described next.

4.2 OpenMP in SPMD Programming Style

Under the SPMD parallelization strategy, we partition the arrays among the threads, and convert the local part into an array that is private to a 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. 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 faster [13].

The SPMD style has been used in conjunction with OpenMP by other researchers (eg. [10, 12]). In [13], the author points out that OpenMP used in this style will outperform MPI communication on ccNUMA machines, and notes that it also reduces false sharing.

For the LBE code, we privatize the region of the array that was associated with a thread by the elementwise block distribution. Space is added to store non-local elements of array f that are updated locally. The contents are then copied to a shared buffer. 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 difficult part when creating the SPMD code. The rest of the computation remains the same as before; however, loop iterations may have been assigned to threads "owning" a datum. Local cache is used efficiently in this version of LBE, and there is less intrusion from the operating system while handling shared variables. Shared buffers are updated only once per iteration. The SGI compiler is able to do a particularly good job of optimizing such "vector" updates, so that data transfer cost is further reduced. The SPMD version of LBE shows a remarkable increase in performance for 16 and 32 processors (Fig. 1 (a)) on the Origin.

In the corresponding TreadMarks code, labeled *private* above, the arrays are privatized to realize this SPMD style. Here too, the modification has led to consistent performance improvements for the LBE code.

5 The Cougar Compiler

The development of efficient parallel applications is a time-consuming, and therefore costly, task that requires considerable expertise. The availability of such expertise cannot be taken for granted. When MPI is used to develop a parallel version of an application, a global program analysis is required to select a suitable parallelization strategy and data decomposition. Regions of the program may need to be rewritten. The application developer must consider the control structure of the code, its data access patterns, dependences and more. If OpenMP is used, the application developer must carefully select and restructure loops that are to be executed in parallel, in order to share work evenly, avoid false sharing of data and make good use of the cache associated with the individual processors. Race conditions must be detected and removed, and barrier synchronizations minimized. If the two programming paradigms are combined, current MPI implementations require the user to ensure that MPI calls are performed in regions executed by a single thread. The restructuring problems asociated with each of the individual paradigms must be identified and solved.

The Cougar Compiler is a prototype software tool being developed at the University of Houston to help an applications developer examine a a Fortran application and convert it to OpenMP. It includes a powerful graphical user interface for displaying both source code and related information in text and graphical format.

The Cougar preprocessor first handles include files, and the compiler then analyses the program and stores information about it; the user may subsequently query the system, or direct it to modify the code. Data flow analyses and data dependence analysis is performed, and the callgraph is constructed. In addition to representations of the structure of the program and its individual units, details of data declarations and dependencies, as well as on the use of global variables and existence of aliases, may be requested. It is possible to track a variable, study a subroutine call, or see a summary of the I/O on a specific device.

On-going work aims to provide explicit support for the task of creating OpenMP programs,

possibly in the presence of MPI constructs, and to optimize the form of the OpenMP code. The latter expects that a program will initially be parallelized under OpenMP using loop-level parallelism, either manually or with tool support. Higher performance can be achieved under the SPMD style; the tool should generate this form automatically from the simpler code if the user additionally supplies directives describing the data distribution and the shadow regions.

5.1 Translation to SMPD Style

Translation from the loop parallel OpenMP code to the SPMD OpenMP version 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. We have begun to implement a set of directives that enables us to automate this translation. It includes the data distributions supplied by the vendors, as well as a general block distribution that may help map the data of some unstructured codes. We distribute data to threads, rather than to processors or memories, so that they can be privatized. Data distributed to a thread is "local" to it. An `ON HOME` directive similar to that of Compaq maps iterations or parallel sections to the thread owning a specified data object. The `SHADOW` directive is borrowed from HPF to specify the extent of non-local data accessed by a thread, whether it is read or written, and used to set up buffers for storing and copying this data.

6 Related Work

Several research [5] and vendor tools already support the creation and debugging of loop-level OpenMP programs. 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 [9]. However, it is very hard for the operating system to determine when to migrate data and current commercial implementations do not perform particularly well. Other extensions to OpenMP have been provided for a variety of reasons, including KAI proposals to extend the range of applicability to the language by including workqueue parallelism, and proposals for explicit hierarchical parallelism, including thread groups, most notably in [2].

7 Conclusions and Future Work

OpenMP [4] 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. However, it cannot 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. 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. Loop iterations are executed on threads for which much of the data is private.

At the University of Houston, we are beginning to develop a source-to-source translator that will accept a modest set of extensions to OpenMP and use them to generate an OpenMP

code written in this style. It is part of the Cougar compiler, that is intended to provide more comprehensive support for OpenMP application development.

Acknowledgements

The authors thank Li-Shi Luo for the provision of the LBE code 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.

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] 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.

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

[5] B. Chapman, J. Merlin, D. Pritchard, F. Bodin, Y. Mevel, T. Sorevik, and L. Hill. Tools for Development of Programs for a Cluster of Shared Memory Multiprocessors. In *PDPTA'99*, Las Vegas, USA, 1999.

[6] 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/.

[7] 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.

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

[9] 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.

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

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

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

[14] 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.