OpenMP on Distributed Memory via Global Arrays*

Lei Huang[a], Barbara Chapman[a], and Ricky A. Kendall[b]

[a]Dept. of Computer Science , University of Houston, Texas.
{leihuang,chapman}@cs.uh.edu

[b]Scalable Computing Laboratory, Ames Laboratory, Iowa.
rickyk@ameslab.gov

This paper discusses a strategy for implementing OpenMP on distributed memory systems that relies on a source-to-source translation from OpenMP to Global Arrays. Global Arrays is a library with routines for managing data that is declared as shared in a user program. It provides a higher level of control over the mapping of such data to the target machine and enables precise specification of the required accesses. We introduce the features of Global Arrays, outline the translation and its challenges and consider how user-level support might enable us to improve this process. Early experiments are discussed. The paper concludes with some ideas for future work to improve the performance of this and other approaches to providing OpenMP on clusters and other distributed memory platforms.

1. Introduction

OpenMP [13] provides a straightforward, high-level programming API for the creation of applications that can exploit the parallelism in widely-available Shared Memory Parallel Systems (SMSs). Although SMSs typically include 2 to 4 CPUs, large machines such as those based on Sun's 6800 architecture may have 100 and more processors. Moreover, OpenMP can be implemented on Distributed Shared Memory systems (DSMs) (e.g. SGI's Origin systems). A basic compilation strategy for OpenMP is relatively simple, as the language expects the application developer to indicate which loops are parallel and therefore data dependence analysis is not required. Given the broad availability of compilers for OpenMP, its applicability to all major parallel programming languages, and the relative ease of parallel programming under this paradigm, it has been rapidly adopted by the community. However, the current language was primarily designed to facilitate programming of modest-sized SMSs, and provides few features for large-scale programming. In fact, current implementations serialize nested parallel OpenMP constructs which can help exploit hierarchical parallelism. Also, OpenMP does not directly support other means of addressing the performance of programs on Non-Uniform Memory Access (NUMA) systems. A more serious problem for the broad deployment of OpenMP code is the fact that OpenMP has thus been provided on shared memory platforms only (including ccNUMA platforms) and OpenMP parallel programs can therefore not be executed on clusters. Given the ease with which clusters can be built today, and the increasing demand for a reduction in the effort required to create parallel code, an efficient implementation of OpenMP on clusters is of great importance to the community.

Efforts to realize OpenMP on Distributed Memory systems (DMSs) such as clusters have to date focused on targeting software Distributed Shared Memory systems (software DSMs), such as TreadMarks[1], OMNI/SCASH[14]. Under this approach, an OpenMP program does not need to be modified for execution on a DMS: the software DSM is responsible for creating the illusion of shared memory

*This work was partially supported by the DOE under contract DE-FC03-01ER25502 and by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23. This work was performed, in part, under the auspices of the U.S. Department of Energy (USDOE) under contract W-7405-ENG-82 at Ames Laboratory, operated by Iowa State University of Science and Technology and funded by the MICS Division of the Office in Advanced Scientific Computing Research at USDOE.

by realizing a shared address space and managing the program data that has been declared to be shared. Although this approach is promising, and work is on-going to improve the use of software DSMs in this context, there are inherent problems with this translation. Foremost among these is the fact that their management of shared data is based upon pages, and that these are regularly updated. Basically, the update is not under the application programmer's control and thus cannot be tuned to optimize the performance of the application.

2. Global Arrays

Compared with MPI programming, Global Arrays (GA)[10] simplifies parallel programming on DMSs by providing users with a conceptual layer of virtual shared memory. Programmers can write their parallel program for clusters as if they have shared memory access, specifying the layout of shared data at a higher level. GA combines the better features of both message-passing and shared memory programming models, leading to both simple coding and efficient execution. It provides a portable interface through which each process is able to independently and asynchronously access the GA distributed data structures, without requiring explicit cooperation by any other process. Moreover, GA programming model also acknowledges the difference of access time between remote memory and local memory and forces the programmer to determine the needed locality for each phase of the computation. By tuning the algorithm to maximize locality, portable high performance is easily obtained. Furthermore, since GA is a library-based approach, the programming model works with most popular language environments: currently bindings are available for FORTRAN, C, C++ and Python.

3. Translation from OpenMP to GA

Global Arrays programs do not require explicit cooperative communication between processes. From a programmer's point of view, they are coding for NUMA (non-uniform memory architecture) shared memory systems. It is possible to automatically translate OpenMP programs into GA because each has the concept of shared data. If the user has taken data locality into account when writing OpenMP code, the benefits will be realized in the corresponding GA code. The translation can give a user advantages of both programming models: straightforward programming and cluster execution.

We have proposed a basic translation strategy of OpenMP to GA in [7] and are working on its implementation in the Open64 compiler [12]. The general approach to translating OpenMP into GA is to generate one procedure for each OpenMP parallel region and declare all shared variables in the OpenMP region to be global arrays in GA. The data distribution is specified. We currently assume a simple block-based mapping of the data. Before shared data is used in an OpenMP construct, it must be fetched into a local copy, also achieved via calls to GA routines; the modified data must then be written back to its "global" location after the computation finishes. GA synchronization routines will replace OpenMP synchronizations. OpenMP synchronization ensures that all computation in the parallel construct has completed; GA synchronization will do the same but will also guarantee that the requisite data movement has completed to properly update the GA data structures.

The translated GA program (Fig. 1) first calls MPI_INIT and then GA_INITIALIZE to initialize memory for distributed array data. The initialization subroutines only need to be called once in the GA program. For dealing with sequential part of OpenMP program, the translation may either attempt to replicate work or may let the master process carry out the sequential parts and broadcast the needed private data at the beginning of parallel part. A replicated approach may lead to more barriers and we have currently chosen the latter. All the sequential parts and subroutines (sub1 in Fig.1(a)) that do not include any OpenMP parallel construct are executed by the Master process only. However, the calls to subroutines (sub2 in Fig.1(a)) that contain OpenMP parallel constructs need to be executed by every process, since GA programs generate a fixed number of processes at the beginning of the program, in contrast to the possibility of forking and joining threads during execution of an OpenMP code. All other processes except the Master process are idle during the sequential part. We intend to consider alternative approaches to deal with the implied performance problems as part of our future work.

Variables specified in an OpenMP private clause can be simply declared as local variables, since

```
Program test
integer a(100,100),j,k
j=2
k=3
call sub1
$OMP PARALLEL SHARED(a)
PRIVATE(j) FIRSTPRIVATE(k)
...
$OMP END PARALLEL
call sub2
end
subroutine sub2
$OMP PARALLEL
...
$OMP END PARALLEL
end
subroutine sub1
...
end
```

(a)

```
Program test
call MPI_INIT()
call ga_initialize()
common /ga_cb1/ g_a
!create global arrays for shared variables
g_a=ga_create(..)
myid=ga_nodeid()
if(myid .eq. 0) then
    j=2
    k=3
    call sub1
endif
call omp_sub1(k,myid)
call sub2
call ga_terminate()
call MPI_FINALIZE(rc)
end
subroutine omp_sub1(k,myid)
integer i, j, k, g_a, myid,a(100,100)
common /ga_cb1/ g_a
if (myid .eq. 0) call ga_brdcst(MT_INT,k,1,0)
!get a local copy of shared variables
call ga_get(g_a,..,a,)
!perform computation
!put modified shared variables back global arrays
call ga_put(g_a,..,a,)
end
```

(b)

Figure 1. OpenMP program(a) and translated GA program (b)

all such variables are private to each process in a GA program by default. Variables specified in an OpenMP firstprivate clause need to be passed as arguments to an OpenMP subroutine (omp_sub1 in Fig.1 (b)), and broadcast to all other processes by the Master process. The translation will turn shared variables into distributed global arrays in GA code by inserting a call to the GA_CREATE routine. The handlers of global arrays need to be defined in a common block so that all subroutines can access the global arrays. GA permits the creation of regular and irregular distributed global arrays. If needed, ghost cells are available. The GA program will make calls to GA_GET to fetch the distributed global data into a local copy. After using this copy in local computations, modified data will be transferred to its global location by calling GA_PUT or GA_ACCUMULATE. GA_TERMINATE and MPI_FINALIZE routines are called to terminate the parallel region.

OpenMP's FIRSTPRIVATE and COPYIN clauses are implemented via the GA broadcast routine GA_BRDCST. The REDUCTION clause is translated by calling GA's reduction routine GA_DGOP. GA library calls GA_NODEID and GA_NNODES are used to get process ID and number of processes, respectively. OpenMP provides routines to dynamically change the number of executing threads at runtime. We do not attempt to translate these since this would amount to redistributing data and GA is based upon the premise that this is not necessary.

In order to implement OpenMP loop worksharing directives, the translated GA program calculates the new lower and upper loop bounds in order to assign work to each CPU based on the specified schedule. Each GA process fetches a partial copy of global data based on the array region read in the local code. Several index translation strategies are possible. A simple one will declare the size of each

local portion of an array to be that of the original shared array; this avoids the need to transform array subscript expressions [16]. For DYNAMIC and GUIDED schedules, the iteration set and therefore also the shared data, must be computed dynamically. In order to do so, we must use GA locking routines to ensure exclusive access to code assigning a piece of work and updating the lower bound of the remaining iteration set; the latter must be shared and visible to every process. However, due to the expense of data transfer in distributed memory systems, DYNAMIC and GUIDED schedules may not be as efficient as static schedules, and may not provide the intended benefits.

The OpenMP SECTION, SINGLE and MASTER directives can be translated into GA by inserting conditionals to ensure that only the specified processes perform the required computation. GA locks and Mutex library calls are used to translate the OpenMP CRITICAL and ATOMIC directives. OpenMP FLUSH is implemented by using GA put and get routines to update shared variables. This could be implemented with the GA_FENCE operations if more explicit control is necessary. The GA_SYNC library call is used to replace OpenMP BARRIER as well as implicit barriers at the end of OpenMP constructs. The only directive that cannot be efficiently translated into equivalent GA routines is OpenMP's ORDERED. We use MPI library calls, MPI_Send and MPI_Recv, to guarantee the execution order of processes if necessary.

Compared with the Software DSM approach for implementing OpenMP on distributed memory system, our strategy uses precise data transfer among processes instead of page-level data migration. Our approach may overcome the overhead of memory coherence and avoid false sharing or redundant data transfer in page-based software DSM. Furthermore, the explicit control mechanisms of OpenMP (or GA) allow the application developer to tune the synchronization events based on the performance and data requirements directly.

4. Performance Issues

An interactive translation from OpenMP to GA could achieve better performance by getting some help from users. For example, we currently assume a block distribution for shared arrays: it is helpful by providing a better data distribution information by users, or using the GA ghost (halo) interface as required by the algorithm. GA supports a variety of block data distribution. However, some applications need cyclic data distribution for better load-balance, which GA does not support.

The SPMD style OpenMP program [5,9] has been investigated by some researchers, which provides better performance on cache coherent Non-Uniform Memroy Access (cc-NUMA) and software DSM systems by systematically appling data privatization. However, The systematically applied data privatization may require more programming effort. Translation from the SPMD style OpenMP to GA program can achieve better performance in DSM system, but this is complex for user to write this sytle of OpenMP program.

Relaxing synchronization achieves better performance by allowing sequential work to be executed while waiting for synchronization. It is also useful in GA programs which have multiple synchronization mechanisms, fences, mutexes, and a global barrier as described above. For example, at the end of an OpenMP parallel loop construct a set of fences that identify the data synchronization required at the next immediate phase of computation could be used instead of a full barrier (GA_SYNC).

5. Benchmarks

We have translated small OpenMP programs into GA and tested their performance and scalability. The experiments shown here compare serial with OpenMP versions of the Jacobi code with a 1152*1152 matrix as input. Fig.2 (a) gives the performance of the Jacobi OpenMP and GA programs on an Itanium 2 cluster with 1 1GHz 4-CPU node and 23 900MHz 2-CPU nodes at the University of Houston; The results in Fig.2 (b) were achieved using an SGI Origin2000 DSM system from NCSA, a hypercube with 128 195MHz MIPS R10000 processors, in multiuser mode. Fig.2 (c) shows performance of this code on a 4*4 SUN cluster (1 4-way ULtraSPARC-II 400 MHz E450 and 3 4-way 450MHz E420s) with Gigabit Ethernet connectivity; The serial version was measured on the E450 machine. The results of our experiments present that a straightforward translation from OpenMP to GA program could achieve good scalability in DSM systems. We expect to get better performance by appling more optimizations.

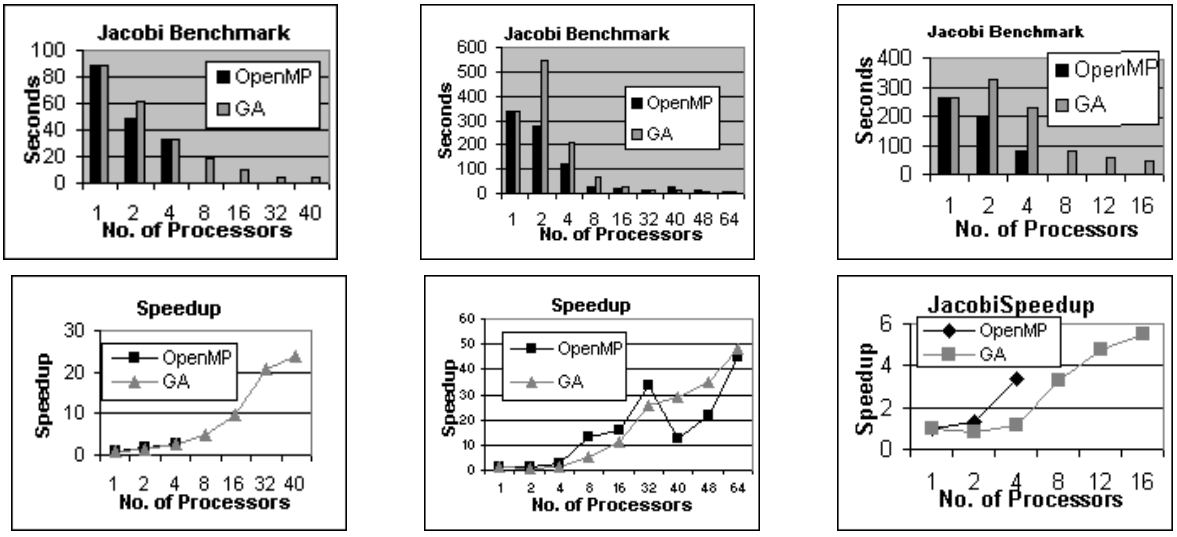

Figure 2. Jacobi program performance on an Itanium2 cluster(a), SGI Origin 2000(b), Sun Cluster(c)

6. Related Work

There have been a variety of efforts that attempt to implement OpenMP on a distributed memory system. Some of these are based upon efforts to provide support for data locality in ccNUMA systems, where mechanisms for user-level page allocation[11], migration, data distribution directives have been developed by SGI [4,15] and Compaq [3]. Data distribution directives can be added to OpenMP [6]. However, this will necessitate a number of additional language changes that do not seem natural in a shared memory model.

A number of efforts have attempted to provide OpenMP on clusters by using it together with a software distributed shared memory (software DSM) environment [1,2,14]. Although this is a promising approach, it does come with high overheads. An additional strategy is to perform an aggressive, possibly global, privatization of data. These issues are discussed in a number of papers, some of which explicitly consider software DSM needs [2,5,9,17].

The approach that is closest to our own is an attempt to translate OpenMP directly to a combination of software DSM and MPI [8]. This work attempts to translate to MPI where this is straightforward, and to a software DSM API elsewhere. While this has similar potential to our own work, GA is a simpler interface and enables a more convenient implementation strategy. GA is ideal in this respect as it retains the concept of shared data.

7. Conclusions

This paper presents a basic compile-time strategy for translating OpenMP programs into GA programs. Our experiments show good scalability of a translated GA program in distributed memory systems, even with relatively slow interconnects. There are several ways to implement OpenMP on clusters. A direct translation such as that proposed here allows precise control of parallelism and creation of a code version that user could manually improve if desired. All cluster code would benefit from a modification that explicitly considers how data/work will be mapped to a machine and this is no different. Some additional user information would also benefit translation, but this could be outside of the OpenMP standard. We intend to explore these issues further as part of our work on an implementation that will enable us to handle large-scale applications.

REFERENCES

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

[2] A. Basumallik, S-J. Min and R. Eigenmann: Towards OpenMP execution on software distributed shared memory systems. Proc. WOMPEI'02, LNCS 2327, Springer Verlag, 2002

[3] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, C. D. Offner: Extending OpenMP For NUMA Machines, Proceedings of Supercomputing 2000, Dallas, Texas, November (2000)

[4] R. Chandra, D.-K. Chen, R. Cox, Maydan, D. E., Nedeljkovic, N., and Anderson, J. M.: Data Distribution Support on Distributed Shared Memory Multiprocessors. Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, Las Vegas, NV, June (1997)

[5] B. Chapman. F. Bregier, A. Patil and A. Prabhakar: Achieving Performance under OpenMP on ccNUMA and Software Distributed Shared Memory Systems. Special Issue of Concurrency Practice and Experience, 14:1-17, 2002

[6] B. Chapman and P. Mehrotra: OpenMP and HPF: Integrating Two Paradigms. Proc. Europar '98, LNCS 1470, Springer Verlag, 65-658, 1998

[7] L. Huang, B. Chapman and R. Kendall: OpenMP for Clusters. Proc. EWOMP'03, Aachen, Germany, September (2003)

[8] R. Eigenmann, J. Hoeflinger, R.H. Kuhn, D. Padua et al.: Is OpenMP for grids? Proc. Workshop on Next-Generation Systems, IPDPS'02, 2002

[9] Z. Liu, B. Chapman, Y. Wen, L. Huang and O. Hernandez: Analyses and Optimizations for the Translation of OpenMP Codes into SPMD Style. Proc. WOMPAT 03, LNCS 2716, 26-41, Springer Verlag, 2003

[10] J. Nieplocha, RJ Harrison, and RJ Littlefield: Global Arrays: A non-uniform memory access programming model for high-performance computers. The Journal of Supercomputing, 10:197-220, 1996

[11] Nikolopoulos, D. S., Papatheodorou, T. S., Polychronopoulos, C. D., Labarta, J., Ayguad, E.: Is Data Distribution Necessary in OpenMP? Proceedings of Supercomputing 2000, Dallas, Texas, November (2000)

[12] Open64 Compiler, http://open64.sourceforge.net/userforum.html

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

[14] M. Sato, H. Harada and Y. Ishikawa: OpenMP compiler for a software distributed shared memory system SCASH. Proc. WOMPAT 2000, San Diego, 2000

[15] Silicon Graphics Inc. MIPSpro 7 FORTRAN 90 Commands and Directives Reference Manual, Chapter 5: Parallel Processing on Origin Series Systems. Documentation number 007-3696-003. http:// techpubs.sgi.com

[16] Mario Soukup: A Source-to-Source OpenMP Compiler, Master Thesis, Department of Electrical and Computer Engineering, University of Toronto

[17] T.H. Weng and B. Chapman: Asynchronous Execution of OpenMP Code. Proc. ICCS 03, LNCS 2660, 667-676, Springer Verlag, 2003