OpenMP for Clusters

Lei Huang*, Barbara Chapman*, Ricky Kendall[+]

*Dept. of Computer Science , University of Houston Texas
[+]Scalable Computing Laboratory, Ames Laboratory, Iowa
{leihuang, chapman}@cs.uh.edu, rickyk@ameslab.gov

Abstract. This paper presents a source-to-source translation strategy from OpenMP to Global Arrays in order to extend OpenMP to distributed memory systems. This translation provides a simple approach for programmers to write parallel programs using a high-level API that will run on both shared memory and distributed memory systems. Our benchmark experiments show scalability and lead us to believe that this approach is more promising than the use of software DSM systems.

1 Introduction

Parallel computer architectures in broad use include Shared Memory Systems (SMSs), Distributed Memory systems (DMSs), and Distributed Shared Memory systems (DSMs). Clusters and DMSs are increasingly popular because of their good price/performance ratio. However, most programs written for them are SPMD (Single Program Multiple Data) programs using the explicit parallel interface MPI (Message-Passing Interface) for communication and synchronization of processes. But MPI requires advanced programming skills, is error prone, and is too complex for some classes of users. Message passing programming models often lead to multiple versions or a complex set of parameters that users must set to get truly portable performance.

Global Arrays (GA) [10] was designed to simplify the programming methodology on distributed memory systems. It provides a portable interface via which processes in an SPMD-style parallel program don't need the explicit cooperation of other processes. The most innovative idea of GA is that it provides an asynchronous one-sided, shared-memory programming environment for distributed memory systems. In contrast to other popular approaches, it does so by providing a library of routines that enable the user to specify and manage access to shared data structures in a program. GA reduces the effort required to write parallel program for clusters since they can

---

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.

assume a virtual shared memory. Part of the task of the user is to explicitly define the physical data locality for the virtual shared memory and the appropriate access patterns of the parallel algorithm.

OpenMP [5] has emerged as a popular parallel programming interface for medium scale high performance applications on SMSs. Strong points are its ability to support incremental parallelization, portability, and ease of use. The OpenMP programmer inserts parallel directives and does not need to change the control flow of the corresponding sequential program, which dramatically reduces the effort of adapting a previously sequential program to parallel form. Although OpenMP is attractive for parallel programming in general, it cannot be used directly on a DMS.

In this paper, we show how Global Arrays may be used to implement OpenMP on clusters. We explain why we believe that this is worthwhile in the next section below, outline the translation from OpenMP to GA in Section 3 and give some benchmarks to show the potential performance of applications translated in this manner on a variety of current platforms. We conclude by briefly discussing related work and future plans.

2 Motivation

Compared with MPI programming, GA 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. However, it does not change the parallel programming model dramatically since programmers still need to write SPMD style parallel code and deal with the complexity of distributed arrays by identifying the specific data movement required for the parallel algorithm. The GA programming model 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.

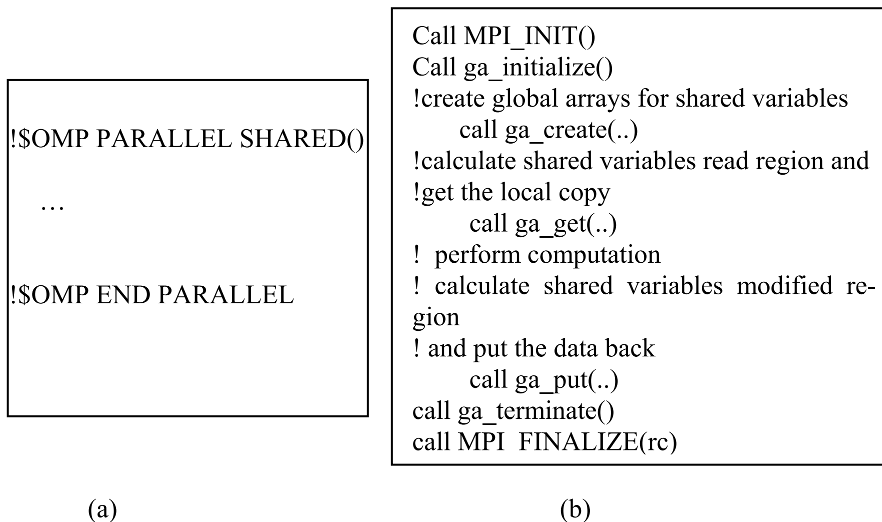
OpenMP provides an efficient and simple parallel programming methodology for SMSs. Given its broad acceptance in the community and the need for a simpler programming model for clusters, we believe that OpenMP should also be adapted to run on clusters, whether via extensions to the standard or improvements in compiler and runtime system technology [8]. However, the traditional approach to doing so requires use of a software DSM to manage shared data in a program. Such systems potentially exchange large amounts of superfluous data at synchronization points in the code, since they transfer pages even when just one element on a page has been updated; thus their ability to provide good performance is unclear. OpenMP programs map computation to threads and hence indirectly specify the data needed by a thread. This attribute makes it possible to translate OpenMP programs into GA programs. 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 the

advantages of both programming models: straightforward programming and cluster execution.

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.

A careful study of OpenMP directives and GA routines showed that almost all OpenMP directives can be translated into GA or MPI library calls at source level. (We may use these together if needed, since GA was designed to work in concert with the message passing environment.) Most of OpenMP library routines and environment variables can be also be translated to GA routines. Exceptions are those that dynamically set/change the number of threads, such as OMP_SET_DYNAMIC, OMP_SET_NUM_THREADS, may not be translated. The general approach to translating OpenMP into GA is to declare all shared variables in the OpenMP program to be global arrays in GA. 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.



(a) (b)

Fig. 1. OpenMP Parallel Region(a) and translated GA program (b)

The translated GA program (cf. Fig. 1) first calls MPI_INIT and then GA_INITIALIZE to initialize memory for distributed array data. These initialization

functions only need be called in GA program. Variables specified in an OpenMP private clause can be simply declared as local variables, since all such variables are private to each process in a GA program by default. The translation will turn shared variables into distributed global arrays in GA code by inserting a call to the GA_CREATE routine. 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 local computations have been performed using this copy, 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.

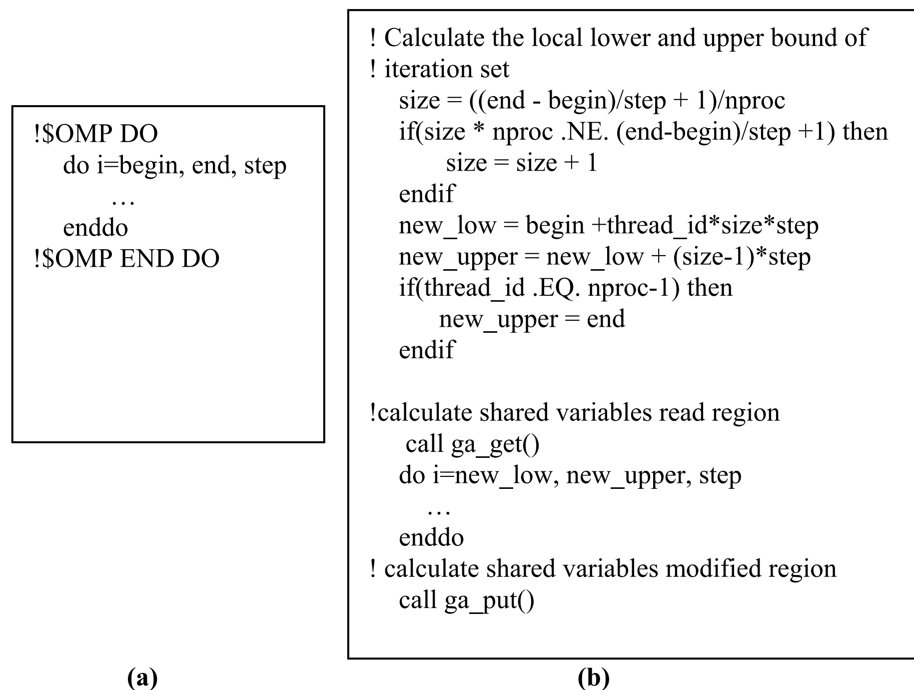


Fig. 2. OpenMP DO construct (a) and translated GA program (b)

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 (e.g. Fig 2 shows the default static block 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 sim-

ple 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 [14]. 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.

```
!$OMP PARALLEL SHARED (a,b,sum)
!$OMP DO
      do j = 2, SIZE
        do i = 2, SIZE
          a(i, j) = (b(i - 1, j) + b(i + 1, j) + b(i, j - 1) + b(i, j + 1)) / 4
        enddo
      enddo
!$OMP END DO
!$OMP DO
      do j = 2, SIZE
        do i = 2, SIZE
          b(i, j) = a(i, j)
        enddo
      enddo
!$OMP END DO
!$OMP DO REDUCTION(+:sum)
      do j = 1,SIZE_1
        do i = 1,SIZE_1
          sum = sum + b(i,j)
        end do
      end do
!$OMP END DO
!$OMP END PARALLEL
```

Fig. 3. Jacobi OpenMP program fragment

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.

Fig. 3 shows a fragment of a simple Jacobi OpenMP program with three shared variables: a, b, sum. To translate this, we need to analyze the access pattern for each shared array in order to minimize inter-process communication in the resulting code. There is no need to declare a global array for the scalar shared variable sum, since it can be handled by the GA reduction function.

```
call MPI_INIT()
call ga_initialize()
myid = ga_nodeid()
nproc = ga_nnodes()
! create Global Arrays for shared variables
OK=ga_create(MT_DBL, SIZE_1, SIZE_1, 'A', SIZE_1, SIZE_1/nproc, g_a)
OK=ga_create(MT_DBL, SIZE_1, SIZE_1, 'B', SIZE_1,SIZE_1/nproc, g_b)
!compute new low bound and upper bound for each thread
psize = ((SIZE - 2) + 1)/nproc
if(psize * nproc .NE. (SIZE-2)+1) then
     psize = psize + 1
endif
new_low = 2 + myid*psize
new_upper = new_low + (psize-1)
if(myid .EQ. nproc-1) then
     new_upper = SIZE
endif
!compute the array read region for each thread
jlo = new_low - 1
jhi = new_upper + 1
!get array local read region
call ga_get(g_b, 1, SIZE_1, jlo, jhi, b(1,jlo), ld)
call ga_sync()
do j = new_low, new_upper
  do i = 2, SIZE
      a(i, j) = (b(i - 1, j) + b(i + 1, j) + b(i, j - 1) +b(i, j + 1)) / 4
 enddo
enddo
! compute array write region for each thread
jlo = new_low
jhi = new_upper
! put array local data back global arrays)
call ga_put(g_a, 2, SIZE, jlo, jhi, a(2, jlo), ld)
call ga_sync()
…..
call ga_terminate()
call MPI_FINALIZE(rc)
```

Fig. 4. Global Array version of Jacobi program

We create two global arrays g_a and g_b (Fig. 4) and distribute them in the j dimension following OpenMP program semantics, which leads to this usage pattern. For each OpenMP DO construct, we compute the new bounds for the chunk to be executed by each thread, and the region of each shared array that is read by a thread. Then we get the local data from the corresponding global array using GA_GET, prior

to executing the loop. After the computation has completed, we compute the modified array region and put the locally written data back into its global storage.

4 Benchmarks

We have translated small OpenMP programs nto GA and tested their performance and scalability. The first three experiments shown here use the Jacobi code and a 1152*1152 matrix; the last set of timings uses this code and a 2304*2304 matrix. Fig. 5 gives the performance of the Jacobi OpenMP and GA programs on an Itanium 2 cluster with 24 900MHz 2-CPU nodes at the University of Houston; each has 4 GB memory. The Scali interconnect has a system bus bandwidth of 6.4GB/s and a memory Bandwidth of 12.8GB/s. The Intel Fortran 7.1 compiler was used with the switches –O2 –i8 –cm –w90 –w95 –align –LINUX64.

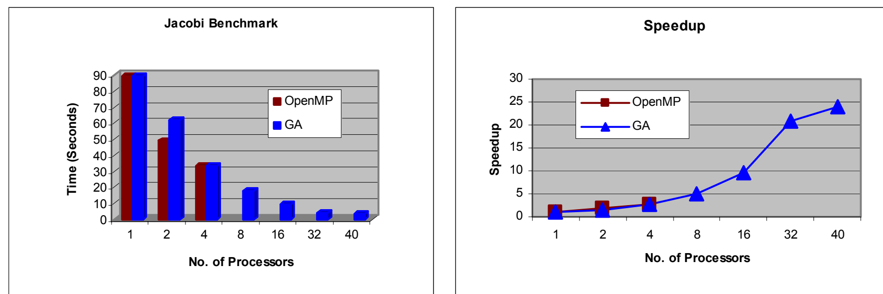


Fig. 5. Jacobi program performance on an Itanium 2 cluster

The results in Fig. 6 were achieved using an SGI Origin2000 DSM system from NCSA, a hypercube with 128 195MHz MIPS R10000 processors, in multiuser mode. The OpenMP performance drops when the Jacobi program uses more than 32 processors as a result of the structure of the interconnect.

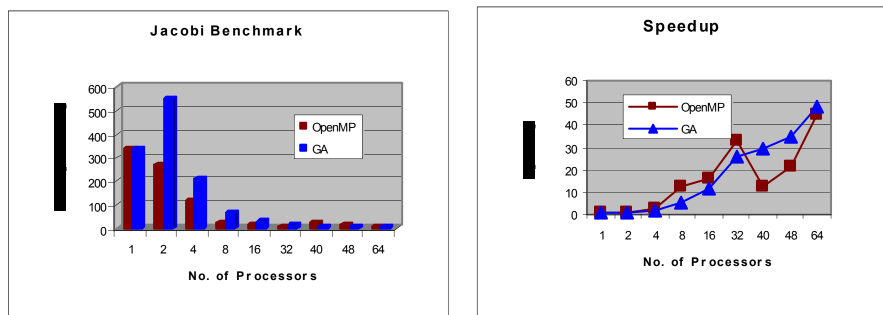


Fig. 6. Jacobi OpenMP and GA performance on SGI 2000

Fig. 7 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 compiler is Sun's Forte Developer release 7.

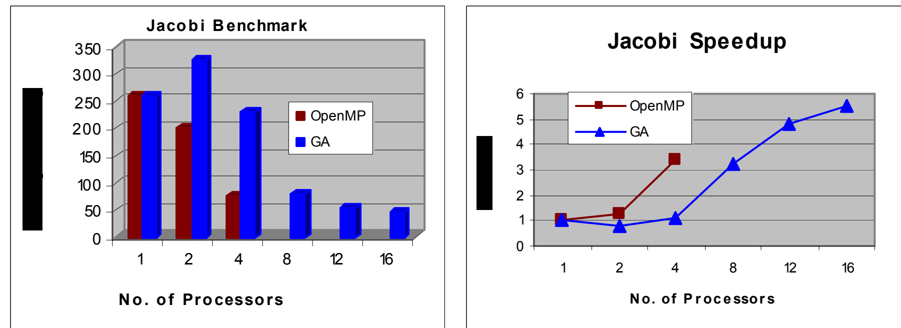


Fig. 7. Jacobi OpenMP and GA performance on a SUN cluster

The last experiment performed was on NERSC's IBM SP RS/6000, a distributed memory machine with 6,080 375 MHz POWER 3+ CPUs with 16GB to 64 GB memory for each node. They are connected to an IBM "Colony" high-speed switch via two "GX Bus Colony" network adapters. We increased the matrix size to 2304*2304 in order to utilize more processors. We have not included results for more than 56 processors since there was not enough computation remaining to keep additional processes busy.

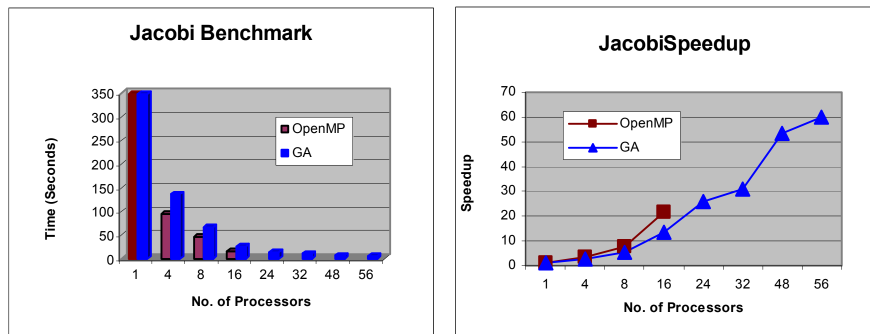


Fig. 8. Jacobi OpenMP and GA programs performance in NERSC IBM SP cluster

5 Related Work

OpenMP is not immediately implementable on distributed memory systems. Given its potential as a high level programming model for large applications, this is a serious

matter and it has clearly been recognized as such by the community. There have been a variety of efforts that attempt to overcome this.

Some of these are based upon efforts to provide support for data locality in ccNUMA systems, where mechanisms for user-level page allocation[11] and migration, and data distribution directives have been developed by SGI [13, 4] and Compaq [3]. Data distribution directives can be added to OpenMP [7]. However, this will necessitate a number of additional language changes that do not seem natural in a shared memory model. Moreover, OpenMP code already contains an implicit data distribution, since work is explicitly assigned to threads, or CPUs. The drawback of the OpenMP approach is that the user is encouraged to ignore locality when assigning work, not that there is no assignment of data and work.

A number of efforts have attempted to provide OpenMP on clusters by using it together with a software distributed shared memory (software DSM) environment [1,12 ,2]. Although this is a promising approach, and work will continue to improve results, it does come with high overheads. In particular, such environments generally move data at the page level and may not be able to restrict data transfers to those objects that truly require it. There are many ways in which this might be improved, including prefetching and forwarding of data, general OpenMP optimizations such as eliminating barriers, and using techniques of automatic data distribution to help carefully place pages of data. An additional approach 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, 9,15,6].

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. The purpose of this hybrid approach is that it tries to avoid the software DSM overheads as far as possible. While this has similar potential to our own work, GA is a simpler interface and enables a more convenient implementation strategy. Because it has a straightforward strategy for allocating data, it can also handle irregular array accesses, which is the main reason for retaining a software DSM in the above work. GA data has a global “home” but it is copied to and from it to perform the computation in regions of code; this is not unlike the OpenMP strategy of focusing on the allocation of work. For both models, this works best if the regions are suitably large. If the user is potentially exposed to the end result of the translation, we feel that they should be shielded as far as possible from the difficulties of distributed memory programming via MPI. GA is ideal in this respect as it retains the concept of shared data.

6 Conclusions and Future Work

This paper presents a basic compile-time strategy for translating OpenMP programs into GA programs. Our experiments show good scalability of translated GA program in distributed memory systems, even with relatively slow interconnects. We do not currently attempt to translate into combined OpenMP-GA as might be appropriate for exploiting shared memory on nodes under OpenMP. We intend to explore this issue and begin 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. Chandra, R., Chen, D.-K., Cox, R., 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. Chapman, B., Bregier, F., Patil, A. and Prabhakar, A.: Achieving High Performance under OpenMP on ccNUMA and Software Distributed Share Memory Systems. Currency and Computation Practice and Experience. Vol. 14, (2002) 1-17
6. 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
7. B. Chapman and P. Mehrotra: OpenMP and HPF: Integrating Two Paradigms. Proc. Europar '98, LNCS 1470, Springer Verlag, 65-658, 1998
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., Ayguadé, E.: Is Data Distribution Necessary in OpenMP? Proceedings of Supercomputing 2000, Dallas, Texas, November (2000)
12. M. Sato, H. Harada and Y. Ishikawa: OpenMP compiler for a software distributed shared memory system SCASH. Proc. WOMPAT 2000, San Diego, 2000
13. 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
14. Mario Soukup: A Source-to-Source OpenMP Compiler, Master Thesis, Department of Electrical and Computer Engineering, University of Toronto
15. T.H. Weng and B. Chapman Asynchronous Execution of OpenMP Code. Proc. ICCS 03, LNCS 2660, 667-676, Springer Verlag, 2003