

# Strategies and Implementation for Translating OpenMP Code for Clusters

Deepak Eachempati, Lei Huang, and Barbara Chapman

Department of Computer Science  
University of Houston  
Houston, TX 77204-3475  
{dreachem, chapman}@cs.uh.edu, lhuang5@mail.uh.edu

**Abstract.** OpenMP is a portable shared memory programming interface that promises high programmer productivity for multithreaded applications. It is designed for small and middle sized shared memory systems. We have developed strategies to extend OpenMP to clusters via compiler translation to a Global Arrays program. In this paper, we describe our implementation of the translation in the Open64 compiler, and we focus on the strategies to improve sequential region translations. Our work is based upon the open source Open64 compiler suite for C, C++, and Fortran90/95.

## 1 Introduction

MPI is still the most popular and successful programming model for clusters. It, however, is error-prone and too complex for most non-experts. OpenMP is a programming model designed for shared memory systems that provides simple syntax to achieve easy-to-use, incremental parallelism, and portability. However, it is not available for distributed memory systems including widely deployed clusters. We believe that it is feasible to use compiler technologies to extend OpenMP to Clusters to alleviate the programming efforts on cluster. We have developed strategies[7][8][12] to implement it via Global Arrays (GA)[15].

GA is a library that provides an asynchronous one-sided, virtual shared memory programming environment for clusters. A GA program consists of a collection of independently executing processes, each of which is able to access data declared to be shared without interfering with other processes. GA enables us to retain the shared memory abstraction, but at the same time makes all communications explicit, thus enabling the compiler to control their location and content. Considerable effort has been put into the efficient implementation of GA's one-sided contiguous and strided communications. Therefore, we can potentially generate GA codes that will execute with high efficiency using our translation strategy. On the other hand, our strategy shares some of the problems associated with the traditional SDSM approach to translating OpenMP for clusters: in particular, the high cost of global synchronization, and the difficulty of translating sequential parts of a program.

The traditional approach to implementing OpenMP on clusters is based upon translating it to software Distributed Shared Memory systems (DSMs), notably TreadMarks[6] and Omni/SCASH[17]. The strategy underlying such systems is to manage

shared memory by migrating pages of data, which unfortunately incurs high overheads. Software DSMs perform expensive data transfers at explicit and implicit barriers of a program, and suffer from false sharing of data at page granularity. They typically impose constraints on the amount of shared memory that can be allocated. But this effectively prevents their applications to large problems. [4] translates OpenMP to a hybrid MPI+software DSM in order to overcome some of the associated performance problems. This is a difficult task, and the software DSM could still be a performance bottleneck. [1] translates OpenMP directly to MPI programs.

In this paper, we discuss our approach and implementation for translating OpenMP programs to Global Arrays, especially we present our strategies for handling sequential regions of OpenMP efficiently. The remainder of this paper is organized as follows. We first describe our approach that translates OpenMP to GA in Section 2, and follow by three strategies for handling the sequential parts of OpenMP programs. We will then present a detailed implementation for the work based on the OpenUH compiler. Related work and conclusions are described in the subsequent sections.

## 2 Our Translation Approach

GA [15] was designed to simplify the programming methodology on distributed memory systems by providing a shared memory abstraction. It does so by providing routines that enable the user to specify and manage access to shared data structures, called global arrays, in a FORTRAN, C, C++ or Python program. GA permits the user to specify block-based data distributions for global arrays, corresponding to HPF BLOCK and GEN\_BLOCK distributions which map the identical length chunk and arbitrary length chunks of data to processes respectively. Global arrays are accordingly mapped to the processors executing the code. Each GA process is able to independently and asynchronously access these distributed data structures via get or put routines.

In principle, translating OpenMP programs into GA programs is not difficult since both of them have the concept of shared data and the GA library features match most OpenMP constructs. 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 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`.

Our translation strategy follows OpenMP semantics and translates an OpenMP code to a SPMD style GA program. OpenMP threads correspond to GA processes: a fixed number of processes are generated and terminated at the beginning and end of the corresponding GA program. Instead of flexible forking and joining threads in OpenMP, the GA programs have to keep the fixed number of processes and manage them from the beginning. OpenMP shared data are translated to global arrays that are distributed among the GA processes; all variables in the GA code that are not global arrays are called private variables. OpenMP private variables are replicated to each GA process. All shared OpenMP arrays are given a data distribution and translated to global arrays.

## 2.1 Strategies for Translating Sequential Regions

All strategies for implementing OpenMP on cluster have a problem with sequential regions. A process executing sequential code may need to read or write to shared data which potentially could be distributed across multiple cluster nodes. A major challenge is to devise an effective compiler/runtime strategy to minimize this communication overhead. A simple approach for handling sequential regions would be to restrict execution of enclosed statements to the master process. However, this presents another problem. If the executing process encounters a parallel region, other processes would need to be available to join in executing it. We refer to this as the control flow problem for translating OpenMP sequential region on clusters. We have identified 3 general translation strategies for resolving this control flow problem, which we discuss below.

**Parallel Control Flow.** For this approach, we translate sequential regions such that only one process (typically the master process) executes assignment statements or I/O operations. However, to ensure every process is available for a parallel region, control flow must be executed by all processes if it encloses a parallel region. While sometimes this can be determined at compile time, procedure calls can make it difficult to statically determine if a control flow construct encloses a parallel region. In the cases a control flow construct is free of enclosed parallel regions or calls to procedures with parallel regions, it need only be executed by a single process. Otherwise, all processes must execute the control flow statement. In a similar way, a call to a procedure must be executed by all processes if there is the potential of encountering a parallel region down that call path.

Another issue is that all processes must identically evaluate the control flow condition expression. This means that any input values required for the condition expression must be consistent across all processes. To handle this, the condition expression is evaluated and written to a temporary variable by the executing single process. Then, all processes execute a collective broadcast operation, which will send the resulting value of the expression to all processes. All processes then use this value to evaluate the control flow condition.

**Redundant Execution Approach.** Another way of tackling the control flow problem in sequential regions is to simply allow every process to redundantly execute the entire sequential region, except for certain I/O operations (e.g. writing to a file). There are multiple potential advantages with this approach. Firstly, it can be implemented with a more straightforward translation than parallel control flow. And in theory, if all shared data is replicated for every process, then a sequential region can be redundantly executed by every process without requiring communication to ensure shared data consistency. And, of course, it would not be necessary to broadcast control flow conditions either.

There are, however, some significant drawbacks to this approach as well. It could be impractical to replicate shared data for every process if the data consisted of very large shared arrays. Often, we would prefer to distribute such arrays across the cluster processes. But if we do this then redundant accesses of such data in sequential regions would require communication to/from every process, a considerable drain on the cluster's network bandwidth. Also, for certain applications with resource-intensive sequential

regions (e.g. graphical interfaces), replicating all sequential code is impractical. For these reasons, we believe parallel control flow is a more suitable translation strategy.

**Idle Process Invocation.** The final strategy we have considered for dealing with this problem is for the master process to invoke idle helper processes to work on parallel regions as they are encountered. In this approach, only the master process executes sequential regions while the rest of the processes remain idle. Furthermore, the compiler outlines all parallel regions in the program to separate procedures, and inserts calls to these procedures where the regions used to be. When the master process encounters a call to an outlined parallel region, it will broadcast an index and any shared data references for the procedure to the rest of the processes. All processes are synchronized just before beginning and exiting execution of the outlined parallel region.

## 2.2 Translation of Parallel Region

Before delving into implementation details, we present a short example to illustrate how we translate an OpenMP parallel region into an SPMD-style code for clusters.

```

1  !$omp parallel shared(a)
2    do k = 1, MAX
3  !$omp do
4    do j = 1, SIZE_Y
5      do i = 1, SIZE_X-1
6        a(i,j) = a(i+1, j) + ...
7      end do
8    end do
9  !$omp end do
10 end do
11 !$omp end parallel

```

(a) Original OpenMP

```

1  ! tell runtime to register 'a' as a shared variable (creates
2  ! corresponding GA)
3  call clustermp_register_var('a', varid, TYP_DBL, SIZE_X, SIZE_Y, 2, A)
4  ! signals to runtime we are entering a parallel region
5  call clustermp_entering_parallel()
6  do k0 = 1, MAX
7  ! runtime calculates loop bounds based on process id
8    jlo = clustermp_getlowerbound(1, SIZE_Y)
9    jhi = clustermp_getupperbound(1, SIZE_Y)
10 ! compiler determines region of array that is read in parallel loop
11 call clustermp_read(IS_LOCAL, a, 'a', varid,2,SIZE_X,jlo,jhi)
12 do j0 = jlo, jhi
13   do i0 = 1, SIZE_X-1
14     if ( clustermp_cond_exec() .eq. 1 ) then
15       a(i0,j0) = a(i0+1,j0) + ...
16     end if
17   end do
18 end do
19 ! compiler determines region of array that is written in parallel loop
20 call clustermp_write(IS_LOCAL, a, 'a', varid,1,SIZE_X-1,jlo,jhi)
21 call clustermp_barrier()
22 end do
23 call clustermp_leaving_parallel()

```

(b) Resulting Parallel Code for Cluster

**Fig. 1.** OpenMP code is translated into an SPMD-style parallel program by our compiler. Calls to the ClusterMP runtime are inserted to manage shared data and coordinate work in sequential and parallel regions.

Fig.1 (a) depicts a simple OpenMP parallel region where work is distributed across the array  $a$ 's second dimension. Suppose the size of  $a$ 's first dimension is  $SIZE\_X$ , and the size of  $a$ 's second dimension is  $SIZE\_Y$ . We determine that  $a$  is a shared variable with work distributed across its second dimension, and we register it as such with our runtime Fig. 1 (b), line 3. This registration will create a corresponding GA for the array  $a$  that is block distributed across each process executing on the cluster. A handle for the GA is managed internally by the runtime, which allows potential runtime optimizations such as redistribution based on dynamic access patterns. Next, we signal the runtime to enter a parallel state, and we commence with the  $MAX$  iterations.

On lines 8 and 9 in Fig. 1 (b), lower and upper bounds for the second dimension are calculated by each process at runtime. On line 11, each process performs a block-wise “get” for array elements it will be reading in the loop. We apply a condition for the assignment on line 14, which restricts execution of the enclosed block of statements depending based on the process id and the OpenMP state (i.e. sequential or parallel region). At the end of the iteration, each process will “commit” the local writes it performed via the *clustermmp\_write* call and then synchronize with the other processes before starting the next iteration. Finally, after all iterations have completed, we tell the runtime that the parallel region has completed.

### 3 Implementation

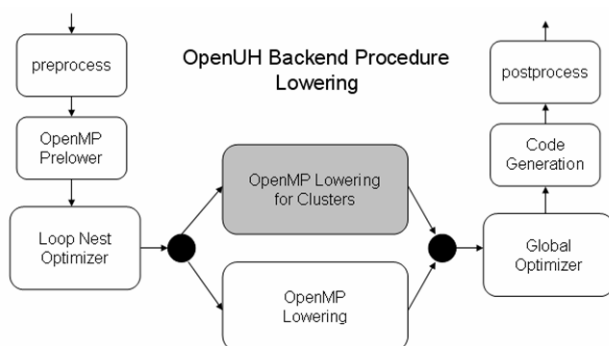
We have implemented the parallel control flow translation strategy for sequential regions, described in the previous section. Our implementation consists of two phases: (1) we use the OpenUH compiler to translate OpenMP programs into a corresponding cluster-enabled parallel program, and (2) we created a runtime library, *ClusterMP*, which wraps calls to the Global Arrays toolkit for creating, reading, and writing shared data in a cluster environment. We present implementation details for the compilation and runtime phases below.

#### 3.1 OpenUH Compiler

The OpenUH compiler is based on Open64, an open source compiler for C, C++, and Fortran 90. Fig. 2 shows a component diagram for the OpenUH backend, where we have implemented our OpenMP translation for execution on clusters.

Our translation approach is to leverage as much as possible the ClusterMP runtime in order to reduce complexity in our OpenMP translation. Consequentially, our translation scheme is fairly straightforward and does not assume an underlying use of global arrays by the runtime. We also feel there is considerable scope for optimizations to be carried out both at the compiler and runtime level. In this section, we describe the translation scheme employed by OpenUH to translate OpenMP programs for execution on a cluster in conjunction with the ClusterMP runtime.

In our translation, each procedure is handled separately. Specifically, if it is the main procedure the compiler will insert a call to initialize the ClusterMP runtime (*clustermmp\_init*) at the beginning of the procedure's body and a call to shutdown the ClusterMP runtime (*clustermmp\_finalize*) at the very end of the procedure's body. For all procedures, a runtime call (*clustermmp\_create\_locals*) is inserted to signal to the



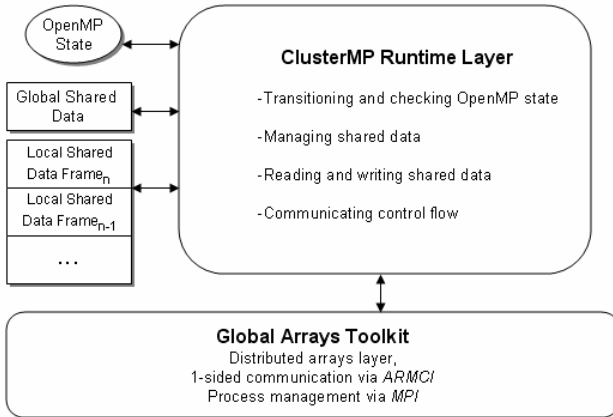
**Fig. 2.** We use the OpenUH compiler to implement the OpenMP translation for clusters

runtime that a new procedure has been entered. This will cause the runtime to push a new shared data frame for the current procedure onto the runtime's shared data frame stack. Then, a runtime call (*clustermmp\_check\_args*) is inserted to communicate the procedure's parameters to the runtime.

The compiler first traverses the IR tree for the procedure and adds any identified shared variables to a shared data list. Calls (*clustermmp\_register\_var*) are inserted for each shared variable to register it with the runtime. Information passed includes array bounds information (all dimension sizes are set to 1 if it is a scalar), the variable name, a unique identification number, and the variable data type. Next, the compiler traverses the IR a second time and restructures the code such that computations in sequential regions are handled by the master process, and all processes participate in the parallel regions as specified by the OpenMP directives. Runtime calls are inserted to read/write shared data, check or modify the dynamic OpenMP state (e.g. in a sequential or parallel region?), and to receive the assigned iterations for OpenMP DO loops. Finally a runtime call (*clustermmp\_pop\_locals*) is inserted at the end of the procedure to signal the runtime to pop the current shared data frame from its shared data frame stack.

### 3.2 ClusterMP Runtime

Considerable research has been carried out for executing OpenMP programs on cluster architectures. We believe the GA toolkit is a suitable target for translating OpenMP programs. It allows for a relatively straightforward translation due to its shared memory abstraction. But unlike software distributed memory systems (SDSMs), which are traditionally used for implementing OpenMP on clusters, GA makes all communication explicit which is useful for controlling and optimizing data communication. For our implementation of OpenMP on clusters we developed a runtime library, ClusterMP, which wraps functionality provided by the GA toolkit and provides management of shared data. In this section, we present an overview of the ClusterMP Runtime implementation.



**Fig. 3.** The runtime core functionality can be split into four categories: (1) transitioning and checking OpenMP state, (2) managing shared data, (3) reading and writing to shared data, and (4) communicating control flow information to all processes

**OpenMP State Transition.** OpenMP allows parallel regions to extend past the lexical extent of the OMP parallel directive and into called procedures. Consequentially, we must maintain dynamic OpenMP state information which indicates whether the executing process is currently in a sequential or parallel region. We add other states to distinguish OpenMP single and critical regions as well. The compiler inserts runtime calls into the translated code to initiate state transition as appropriate. All processes, by default, start in the sequential state. When transitioning from sequential to parallel (via *clustermp\_entering\_parallel*) or from parallel to sequential (via *clustermp\_leaving\_parallel*), the runtime will implicitly invoke a barrier operation to synchronize all the processes, unless a *nowait* clause was specified.

The value of retaining OpenMP state for the purposes of our translation strategy is to identify whether a process may execute a conditional block. As explained in our description of the compiler translation, we place all assignment statements (i.e. computation) in a conditional block. The condition is evaluated to true at runtime simply if the process ID is 0 (it is the master process) or the OpenMP state is set to parallel or critical. This will restrict other processes from performing computation in sequential regions or OpenMP single blocks.

**Managing Shared Data.** The runtime uses two data structures for maintaining metadata corresponding to all shared variables identified for the running process. The *Global Shared Data* table holds metadata for shared global variables. The *Local Shared Data Frame (LSDF)* stack holds metadata for all shared variables local to each procedure in the call stack. The shared data we support may be either a scalar or multi-dimensional array with a base data type of integer, floating point, or complex number. For each shared variable, we track the variable name, a unique variable id, a global array handle, and the number of dimensions and the size of each dimension (1 and 1 for scalars).

We use global arrays to represent shared data. Whenever `clustermmp_register_var` is invoked for a shared variable, the ClusterMP runtime will attempt to create a corresponding global array for the variable with an appropriate data distribution. Information for this global array is then added to the appropriate Shared Data table. When a process enters a called procedure, `clustermmp_create_locals` is called to create a new *LSDF* that records information for shared variables encountered in the procedure. This frame is pushed onto a stack which corresponds to the call stack for the process. Global array information for parameters passed into a procedure can be copied from the previous *LSDF* into the new frame, using the calls `clustermmp_set_args` and `clustermmp_check_args`.

**Reading and Writing to Shared Data.** The compiler inserts a call to `clustermmp_read` whenever data that is not explicitly scoped as private is read, and it inserts a call to `clustermmp_write` whenever such data is written to. The `clustermmp_read` operation will perform a one-sided get on the corresponding global array for the variable being read, and copy the retrieved value into the variable address (which is essentially treated as a “local buffer” for the global array). Similarly, `clustermmp_write` will perform a one-sided put on the corresponding global array using the value provided in the variable. If either function is invoked for a variable that hasn’t been registered, then it immediately returns. The steps taken for reading and writing shared data is identical for sequential and parallel regions.

Clearly, this approach incurs very high overhead, simply due to the fact that a runtime call is made for each variable access. This means that it is the job of the runtime to minimize communication overhead as much as possible. We have investigated various strategies for minimizing communication – including prefetching read data and delaying writes to shared data until a synchronizing barrier is reached. We believe there is considerable scope for optimizing the overhead of reading or writing shared data, both in the compiler and the runtime.

**Communicating Control Flow.** In our translation, the master process must communicate the resulting value of the Boolean expression for control flow constructs. The routine `clustermmp_comm_cf` is used to communicate this information. If the OpenMP state is not sequential or single, then this routine does nothing (all processes already have the necessary information to correctly execute the control flow). Otherwise, we broadcast the value specified from the master process to all other processes.

## 4 Evaluation

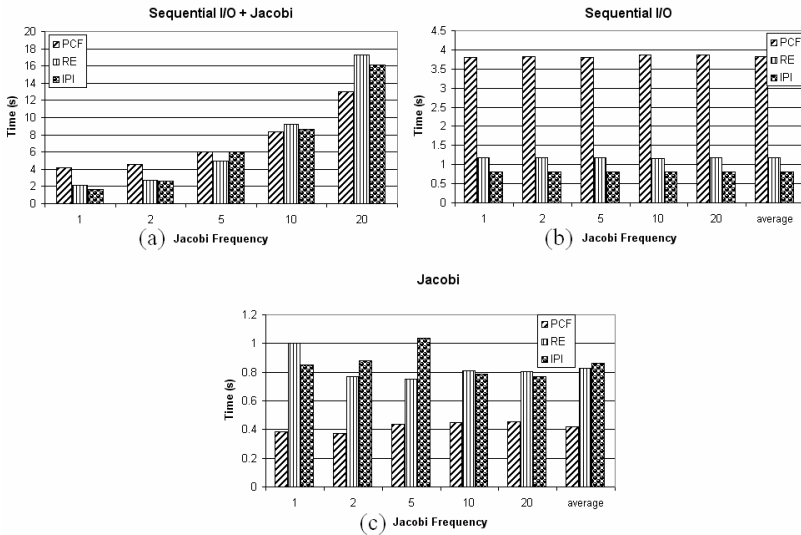
In this section, we discuss the results of a set of experiments we ran to evaluate the three strategies for sequential translation described in Section 2.1. We look at execution time and frequency of various GA routine calls. Experiments were conducted on *Atlantis*, a cluster of dual-Itanium2 machines connected by a high performance myrinet interconnect. We used OpenUH 1.0 to implement the parallel control flow (PCF) translation and we create *ClusterMP* programs by hand to test redundant execution (RE) and idle process invocation (IPI). These experiments also used the GA 4.0.5 and Intel OpenMPI 1.2.3 libraries.

### 4.1 Benchmark

To test our implementation, we devised a benchmark which would contain a configurable mixture of sequential regions, with read and write I/O, and parallel regions. Within the main iteration loop of the benchmark: (1) the contents of a large file are read into a buffer, (2) an arithmetic expression is evaluated based on the iteration count and one of multiple procedures are called accordingly. (3) a parallel routine (*Jacobi*) may or may not be called (likelihood increases proportionally with a specified *JacobiFrequency* parameter), and (4) contents of file are written back out to an output file. We ran this benchmark, translated using the above three strategies, on 4 cluster nodes.

### 4.2 Performance Results

The results in Fig. 4 (a) show that the PCF approach yielded the best total execution time as the frequency of Jacobi interjection increases. While from Fig. 4 (b) it is clear that sequential execution time was much greater for PCF than either RE or IPI, we see also that each invocation of Jacobi is executed faster ( Fig. 4 (c) ). This is an interesting result, since all three approach should yield identical translations for *parallel* regions, and indeed we verified that all three translations of Jacobi had an identical number of calls to *ga\_create*, *ga\_get*, *ga\_put*, *ga\_sync*, and *ga\_destroy*. When worker processes are expected to replicate execution (as in RE) in sequential regions, this can cause them to arrive late at the parallel region which in turn results in extra wait time to complete necessary collective operations. Thus, even though the RE



**Fig. 4.** The above the three graphs show execution time as the frequency of Jacobi invoked during the run varies. In (a), we look at total execution time of benchmark, in (b) we look only at the “sequential” portion, and in (c) we focus on time for each invocation of Jacobi.

approach can yield superior sequential execution time (*for the master process*), overall performance can be potentially be degraded.

We are particularly intrigued by the results of the IPI approach, which in effect mimics the fork-join model of traditional shared memory programs. Sequential execution times are very small, as expected, since only the master process executes it. As with RE though, there are unexpected overheads when executing the parallel regions that are currently being investigated.

## 5 Related Work

Our approach to implementing OpenMP for distributed memory systems has a number of features in common with approaches that translate OpenMP to Software DSMs [17][2][6][5] or Software DSM plus MPI [4], or directly to MPI [1] for cluster execution. All of these methods need to determine the data that has to be distributed across the system, and must adopt a strategy for doing so. Also, the work distribution for parallel and sequential regions has to be implemented, and it is typically the latter that leads to problems. Note that it is particularly helpful to perform an SPMD-style, global privatization of OpenMP shared arrays before translating codes via any strategy for cluster execution, due to the inherent benefits of reducing the size and number of shared data structures and of obtaining a large fraction of references to (local) private variables.

On the other hand, our translation to GA is distinct from other approaches in that ours promises higher levels of efficiency via the construction of precise communication sets. The difficulty of the translation itself lies somewhere between the translation to MPI and the translation to Software DSMs. First, the shared memory abstraction is supported by GA and Software DSMs, but is not present in MPI. It enables a consistent view of variables and a non-local datum is accessible if given a global index. In contrast, only the local portion of the original data can be seen by each process in MPI. Therefore manipulating non-local variables in MPI is inefficient since the owner process and the local indices of arrays have to be calculated. Furthermore, our GA approach is portable, scalable and does not impose limitations on the shared memory space. The everything-shared SDSM as presented in [3] attempts to overcome the relaxation of the coherence semantics and the limitation of the shared areas in other SDSMs. It does solve commonly existing portability problems in SDSMs by using an OpenMP run-time approach, but it is hard for such a SDSM to scale with sequential consistency. Second, the non-blocking and blocking one-sided communication mechanisms offered in GA allow for flexible and efficient programming. In MPI-1, both sender process and receiver process must be involved in the communication. Care must be taken with the ordering of communications in order to avoid deadlocks. Instead, get and/or put operations can be handled within a single process in GA.

## 6 Conclusion

Despite the widespread use multicores, clusters are increasingly used as compute platforms for a growing variety of applications. Extending the simple programming

model OpenMP to clusters will provide an efficient program paradigm to allow many non-experts to exploit the capacity of clusters. OpenMP is designed for shared memory systems, and it seems inappropriate to add new language features for cluster support into it. A joined compiler and runtime system is a more practical and promising approach for bridging the gap between the language and hardware.

This paper describes our strategies in implementing OpenMP on clusters by translating OpenMP codes to equivalent GA ones. This approach has the benefit of being relatively straightforward. This strategy has the advantage of relative simplicity together with reasonable performance, without adding complexity to OpenMP for both SMP and non-SMP systems. We have presented the strategies for efficiently handling sequential regions in an OpenMP program. We describe our implementation for the translation from OpenMP to GA in the Open64 compiler [16], an open source compiler that supports OpenMP and which we have enhanced in a number of ways already. We have developed a runtime system on top of the GA library for facilitating the compiler implementation. The rich set of analyses and optimizations in Open64 may help us create efficient GA codes. We will continue working on optimizing our translation by utilizing existing compiler analyses. We plan to build a framework to conduct parallel data flow analysis for OpenMP to improve the performance.

## References

1. Basumallik, A., Eigenmann, R.: Towards Automatic Translation of OpenMP to MPI. In: ICS 2005: Proceedings of the 19th Annual International Conference on Supercomputing, pp. 189–198. ACM Press, New York (2005)
2. Basumallik, A., Min, S.-J., Eigenmann, R.: Towards OpenMP Execution on Software Distributed Shared Memory Systems. In: Proceedings of the 4th International Symposium on High Performance Computing, pp. 457–468. Springer, Heidelberg (2002)
3. Costa, J.J., Cortes, T., Martorell, X., Ayguade, E., Labarta, J.: Running OpenMP Applications Efficiently on an Everything-Shared SDSM. In: IPDPS 2004, IEEE Computer Society Press, Los Alamitos (2004)
4. Eigenmann, R., Hoeflinger, J., Kuhn, R.H., Padua, D., Basumallik, A., Min, S.-J., Zhu, J.: Is OpenMP for Grids? In: IPDPS, Fort Lauderdale, FL (April 2002)
5. Hoeflinger, J.P.: Extending OpenMP to Clusters. Technical report. Intel Corporation (2006)
6. Hu, Y.C., Lu, H., Cox, A.L., Zwaenepoel, W.: OpenMP for Networks of SMPs. *Journal of Parallel Distributed Computing* 60, 1512–1530 (2000)
7. Huang, L., Chapman, B., Liu, Z.: Towards a More Efficient Implementation of OpenMP for Clusters via Translation to Global Arrays. *Parallel Computing* 31(10-12) (2005)
8. Huang, L., Chapman, B., Kendall, R.: OpenMP for Clusters. In: The Fifth European Workshop on OpenMP, EWOMP 2003. Aachen, Germany (2003)
9. Kee, Y.-S., Kim, J.-S., Ha, S.: ParADE: An OpenMP Programming Environment for SMP Cluster Systems. In: Proceedings of the ACM/IEEE Supercomputing 2003 Conference. Phoenix (2003)
10. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: An Optimizing, Portable OpenMP Compiler. *Concurrency and Computation: Practice and Experience*, Special Issue on CPC selected papers (2006)

11. Liu, Z., Chapman, B., Wen, Y., Huang, L., Hernandez, O.: Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 26–41. Springer, Heidelberg (2003)
12. Liu, Z., Huang, L., Chapman, B., Weng, T.-H.: Efficient Implementation of OpenMP for Clusters with Implicit Data Distribution. In: Chapman, B.M. (ed.) WOMPAT 2004. LNCS, vol. 3349, pp. 121–136. Springer, Heidelberg (2005)
13. Matsuba, H., Ishikawa, Y.: OpenMP on the FDSM Software Distributed Shared Memory. In: The Fifth European Workshop on OpenMP, EWOMP 2003. Aachen, Germany (September 2003)
14. Nieplocha, J., Carpenter, B., ARMCI,: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. In: Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, pp. 533–546. Springer, Heidelberg (1999)
15. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *Journal of Supercomputing* 10, 169–189 (1997)
16. Open64 Compiler and Tools (November 2003), <http://open64.sourceforge.net/>
17. Sato, M., Harada, H., Hasegawa, A., Ishikawa, Y.: Cluster-Enabled OpenMP: An OpenMP Compiler for SCASH Software Distributed Share Memory System. *Scientific Programming, Special Issue: OpenMP* 9(2,3), 123–130 (2001)