# Efficient Implementation of OpenMP for Clusters with Implicit Data Distribution

Zhenying Liu[1], Lei Huang[1], Barbara Chapman[1], Tien-Hsiung Weng[2]

1. Department of Computer Science, University of Houston
{zliu, leihuang, chapman}@cs.uh.edu

2. Department of Computers Science and Information Management, Providence University
thweng@pu.edu.tw

**Abstract.** This paper discusses an approach to implement OpenMP on clusters by translating it to Global Arrays (GA). The basic translation strategy from OpenMP to GA is described. GA requires a data distribution; we do not expect the user to supply this; rather, we show how we perform data distribution and work distribution according to OpenMP static loop scheduling. An inspector-executor strategy is employed for irregular applications in order to gather information on accesses to potentially non-local data, group non-local data transfers and overlap communications with local computations. Furthermore, a new directive INVARIANT is proposed to provide information about the dynamic scope of data access patterns. This directive can help us generate efficient codes for irregular applications using the inspector-executor approach. Our experiments show promising results for the corresponding regular and irregular GA codes.

## 1 Introduction

A recent survey of hardware vendors [7] showed that they expect the trend toward use of clustered SMPs for HPC to continue; systems of this kind already dominate the Top500 list [20]. The importance of a parallel programming API for clusters that facilitates programmer productivity is increasingly recognized. Although MPI is a de facto standard for clusters, it is error-prone and too complex for non-experts. OpenMP is a programming model designed for shared memory systems that does emphasize usability, and we believe it can be extended to clusters as well.

The traditional approach to implementing OpenMP on clusters is based upon translating it to software Distributed Shared Memory systems (DSMs), notably TreadMarks [9] and Omni/SCASH [18]. Unfortunately, the management of shared data based on pages 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. This effectively prevents their applications to large problems. [6]

This work was supported by the DOE under contract DE-FC03-01ER25502.

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. In contrast, we propose a translation from OpenMP to Global Arrays (GA) [10]. 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.

The translation requires a selection of a data distribution. This is a major challenge, since OpenMP lacks data distribution support. We also need to take data locality and load balancing problems into account when we decide how to distribute the computation to the GA processes. In this paper, we discuss a translation of OpenMP to GA that transparently uses the simple block-based data distributions provided by GA, and discuss additional user information needed for irregular access applications.

The remainder of this paper is organized as follows. We first give an overview of the translation from OpenMP to GA. In section 3, we discuss language extensions and compiler strategies that are needed to implement OpenMP on clusters. Experiments, related work and conclusions are described in the subsequent sections.

## 2 Translation from OpenMP to GA

In this section, we show our translation from OpenMP to GA [10], and describe the support for data distribution in GA. The appropriate work distributions are also discussed.

### 2.1 A Basic Translation to GA

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 g*lobal 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.

It is largely straightforward to translate OpenMP programs into GA programs because both have the concept of shared data and the GA library features match most OpenMP constructs. However, before the translation occurs, we may perform global privatization and transform OpenMP into the so-called SPMD style [13] in order to improve data locality. If each OpenMP thread consistently accesses a region of a shared array, the array may be privatized by creating a private data structure per thread, corresponding to the region it accesses. New shared data structures may need to be inserted to act as buffers, so that elements of the original shared array may be exchanged between threads as necessary. Our translation [10] strategy follows OpenMP

semantics. OpenMP threads correspond to GA processes and 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 will be translated to GA private variables as will some OpenMP shared data. OpenMP scalars and private variables are replicated to each GA process. Small or constant shared arrays in OpenMP will also be replicated; all other shared OpenMP arrays must be given a data distribution and they will be translated to *global arrays*.

Shared pointers in OpenMP must also be distributed. If a pointer points to a shared array, we need to analyze whether the contents of the current pointer is within the local portion of the shared memory. Otherwise, *get* or *put* operations are required to fetch and store non-local data and the pointer will need to point to the local copy of the data. If a shared pointer is used directly, we need to substitute it by an array and distribute the array according to the loop schedule because GA does not support the distribution of C pointers.

Our basic translation strategy assigns loop iterations to each process according to OpenMP static loop schedules. For this, we must calculate the iteration sets of the original OpenMP parallel loops for each thread. Furthermore, we compute the regions of shared arrays accessed by each OpenMP thread when performing its assigned iterations. After this analysis, we determine a block-based data distribution and insert the corresponding declarations of *global arrays*. The loop bounds of each parallel loop are modified so that each process will work on its local iterations. Elements of *global arrays* may only be accessed via *get* and *put* routines. Prior to computation, the required *global array* elements are gathered into local copies. If the local copies of *global arrays* are modified during the subsequent computation, they must be written back to their unique "global" location after the loop has completed. GA synchronization routines replace OpenMP synchronization to ensure that all computation as well as the communication to update global data have completed before work proceeds.

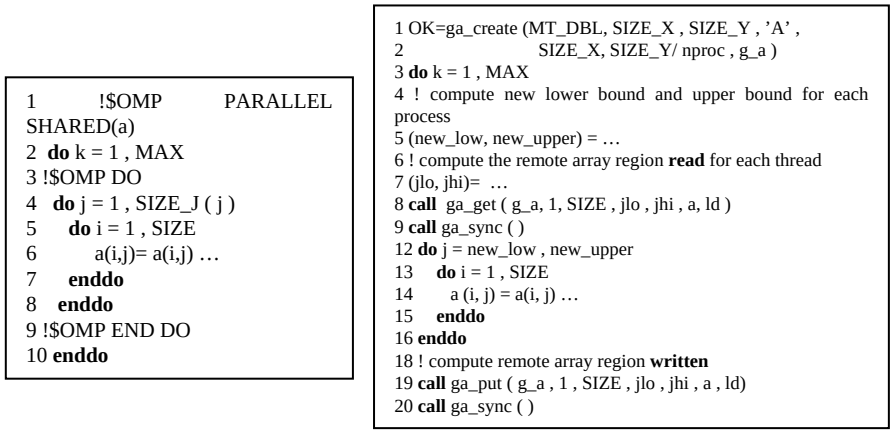


**Fig. 1.**(a) An OpenMP program and (b) the corresponding GA Program

We show an example of an OpenMP program in Fig. 1(a) and its corresponding GA program in Fig. 1(b). The resulting code computes iteration sets based on the process ID. Here, array *A* has been given a block distribution in the second dimension, so

that each processor is assigned a contiguous set of columns. Non-local elements of *global array A* in Fig. 1 (b) are fetched using a *get* operation followed by synchronization. The loop bounds are replaced with local ones. Afterwards, the non-local array elements of *A* are put back via a *put* operation with synchronization.

Unfortunately, the translation of synchronization constructs (CRITICAL, ATOMIC, and ORDERED) and sequential program sections (serial regions outside parallel regions, OpenMP SINGLE and MASTER) may become nontrivial. When translating into GA programs, we will insert synchronization among GA processes. GA has several features for synchronization; however, the translated codes may not be efficient.

### 2.2 Implementing Sequential Regions

We use several different strategies to translate the statements enclosed within a sequential region of OpenMP code including I/O operations, control flow constructs (IF, GOTO, and DO loops), procedure calls, and assignment statements. A straightforward translation of sequential sections would be to use exclusive master process execution, which is suitable for some constructs including I/O operations. Although parallel I/O is permitted in GA, it is a challenge to transform OpenMP sequential I/O into GA parallel I/O. The control flow in a sequential region must be executed by all the processes if the control flow constructs enclose or are enclosed by any parallel regions. Similarly, all the processes must execute a procedure call if the procedure contains parallel regions, either directly or indirectly. We categorize the different GA execution strategies for an assignment statement in sequential parts of an OpenMP program based on the properties of data involved:

1. If a statement writes to a variable that will be translated to a *GA private* variable, this statement is executed redundantly by each process in a GA program; each process may fetch the remote data that it will read before executing the statement. A redundant computation can remove the requirement of broadcasting results after updating a GA private variable.
2. If a statement writes to an element of an array that will be translated to a *global array* in GA (e.g. S[i]=…), this statement is executed by a single process. If possible, the process that owns the shared data performs the computation. The result needs to be put back to the "global" memory location.

Data dependences need to be maintained when a *global array* is read and written by different processes. Our strategy is to insert synchronization after each write operation to g*lobal arrays* during the translation stage; at the code optimization stage, we may remove redundant *get* or *put* operations, and aggregate the communications for neighboring data if possible.

### 2.3 Data and Work Distribution in GA

GA only provides simple block-based data distributions and supplies features to make them as efficient as possible. There are no means for explicit data redistribution. GA's asynchronous one-sided communication transfers the required array elements,

rather than pages of data, and it is optimized to support the transfer of sets of contiguous or strided data, which are typical for HPC applications. These provide performance benefits over software DSMs. With block distributions, it is easy to determine the location of an arbitrary array element. However, since these distributions may not provide maximum data locality, they may increase the amount of data that needs to be gathered and scattered before and after execution of a code region respectively. In practice, this tends to work well if there is sufficient computation in such code regions. In our translation, GA only requires us to calculate the regions of *global arrays* that are read or written by a process to complete the communication; GA handles the other details. It is fast and easy for GA to compute the location of any global data element. We may optimize communication by minimizing the number of *get/put* operations and by grouping small messages into bigger ones.

A user has to determine and specify the distribution of global data in a GA program; thus our translation process must decide on the appropriate block-based data distributions when converting OpenMP programs to the corresponding GA ones. We determine data distributions for a GA program based upon the following simple rules:

1. If most loop index variables in those loop levels immediately enclosed by PARALLEL DO directives sweep over the same dimension of a shared array in an OpenMP program, we perform a one-dimensional distribution for the corresponding array in this dimension;
2. If different dimensions of a shared array are swept over almost evenly by parallel loops, we may perform multi-dimensional distribution for this array.
3. If parallel loops always work on a subset of a shared array, we may distribute this shared array using a GEN_BLOCK distribution; otherwise, a BLOCK distribution may be deployed. In the former case, the working subset of the shared array is distributed evenly to each thread; the first and last thread will be assigned any remaining elements of arrays at the start and end, respectively.

We believe that an interactive tool could collaborate with the user to improve this translation in many cases. One improvement is to perform data distribution based on the most time-consuming parallel loops. Statically, we may estimate the importance of loops. However, user information or profile results, even if based on a partial execution, is likely to prove much more reliable. For example, if an application contains many parallel loops, user information about which ones are the most time-consuming can help us determine the data distribution based upon these specified parallel loops only. We are exploring ways to automate the instrumentation and partial execution of a code with feedback directly to the compiler: such support might eliminate the need for additional sources of information.

Note that it is possible to implement all forms of OpenMP loop schedule including OpenMP static, dynamic and guided loop scheduling. OpenMP static loop scheduling distributes iterations evenly. When the iterations of a parallel loop have different amount of work, dynamic and guided loop scheduling can be deployed to balance the workload. We can perform the work assignment in GA corresponding to dynamic and guided loop scheduling; however, the equivalent GA program may have unacceptable overheads, as it may contain many *get* and *put* operations transferring small amounts of data**.** Other work distribution strategies need to be explored that take data locality and load balancing into account.

In the case of irregular applications, it may be necessary to gather information on the *global array* elements needed by a process; whenever indirect accesses are made to a *global array*, the elements required in order to perform its set of loop iterations cannot be computed. Rather, a so-called inspector-executor strategy is needed to analyze the indirect references at run time and then fetch the data required. The resulting data sets need to be merged to minimize the number of required *get* operations. We enforce static scheduling and override the user-given scheduling for OpenMP parallel loops that include indirect accesses. The efficiency of the inspector-executor implementation is critical. In a GA program, each process can determine the location of data read/written independently and can fetch it asynchronously. This feature may substantially reduce the inspector overhead compared with a message passing program or with a paradigm that provides a broader set of data distributions. Our inspector-executor implementation distributes the loop iterations evenly to processes, assigning each process a contiguous chunk of loop iterations. Then each process independently executes an inspector loop to determine the *global array* elements (including local and remote data) needed for its iteration set. The asynchronous communication can be overlapped with local computations, if any.

### 2.4 Irregular Computation Case Study

We studied one of the programs in the FIRE™ Benchmarks [1]. FIRE™ is a fully interactive fluid dynamics package for computing compressible and incompressible turbulent fluid. *gccg* is a parallelizable solver in the FIRE package that uses orthomin and diagonal scaling. *gccg* which contains irregular data accesses is explored in order to understand how to translate such codes to GA.

```
!$OMP PARALLEL
          DO I = 1, iter
!$OMP DO
          DO 10 NC=NINTCI,NINTCF
             DIREC1(NC)=DIREC1(NC)+RESVEC(NC)*CGUP(NC)
      10 CONTINUE
!$OMP END DO
!$OMP DO
           DO 4 NC=NINTCI,NINTCF
              DIREC2(NC)=BP(NC)*DIREC1(NC)
         X          - BS(NC)  * DIREC1(LCC(1,NC))
         X          - BW(NC) * DIREC1(LCC(4,NC))
         X          - BL(NC)  * DIREC1(LCC(5,NC))
         X          - BN(NC)  * DIREC1(LCC(3,NC))
         X          - BE(NC)  * DIREC1(LCC(2,NC))
         X          - BH(NC) * DIREC1(LCC(6,NC))
       4  CONTINUE
!$OMP END DO
           END DO
!$OMP END PARALLEL
```

**Fig. 2.** An OpenMP code segment in *gccg* with irregular data accesses

We show the process of translation of OpenMP *gccg* program into the corresponding GA program. Fig. 2 displays the most time-consuming part of the *gccg* program. In our approach, we perform array region analysis to determine how to handle the shared arrays in OpenMP. Shared arrays *BP*, *BS*, *BW*, *BL*, *BN*, *BE*, *BH*, *DIREC2* and *LCC* are privatized during the initial compiler optimization to improve locality of OpenMP codes, since each thread performs work only on an individual region of these shared arrays. In the subsequent translation, they will be replaced by GA private variables. In order to reduce the overhead of the conversion between global and local indices, we may preserve the global indices for the list of arrays above when declaring them and allocate the memory for array regions per process dynamically if the number of processes is not a constant. Shared array *DIREC1* is distributed via g*lobal arrays* according to the work distribution in the two parallel do loops in Fig. 2. A subset of array *DIREC1* is swept by all the threads in the first parallel loop; the second parallel loop accesses *DIREC1* indirectly via *LCC*. We distribute *DIREC1* using a GEN_BLOCK distribution according to the static loop schedule in the first parallel loop in order to maximize data locality, as there is no optimal solution of data distribution for *DIREC1* in the second loop. Array region *DIREC1*[*NINTC1*:*NINTCF*] is mapped to each process evenly in order to balance the work. Since *DIREC1* is declared as [1:*N*], the array regions [1:*NINTC1*] and [*NINTCF*:*N*] must be distributed as well. We distribute these two regions to process 0 and the last process for contiguity. Therefore, it is not an even distribution and a GEN_BLOCK distribution is employed as shown in Fig. 3, assuming four processors are involved.

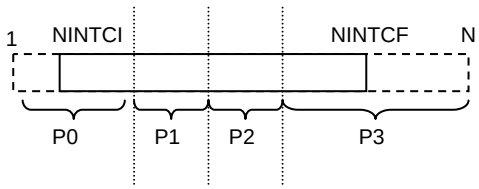


**Fig. 3.** GEN_BLOCK distribution for array DIREC1

As before, we perform work distribution according to OpenMP loop scheduling. In the case that all data accesses will be local the first loop of Fig. 2, loop iterations are divided evenly among all the threads and data are referenced contiguously. If a thread reads or writes some non-local data in a parallel loop, array region analysis enables us to calculate the contiguous non-local data for regular accesses and we can fetch all the non-local data within one communication before these data are read or written. Fortunately, we do not need to communicate for the first loop in Fig. 2 due to completely local accesses. But in the second loop of Fig. 2, some data accesses are indirect and thus actual data references are unknown at compile time. Therefore we cannot generate efficient communications based upon static compiler analysis, and at least one communication per iterations has to be inserted. This would incur very high overheads. Hence, inspector-executor strategy [11] is employed to overcome them.

The inspector-executor approach [17] generates an extra inspector loop preceding the actual computational loop. Our inspector is a parallel loop as shown in Fig. 4. We detect the values for each indirection array in the allocated iterations of each GA process. We use a hash table to save the indices of non-local accesses, and generate a list of

communications for remote array regions. Each element in the hash table represents a region of a *global array*, which is the minimum unit of communication. Using a hash table can remove duplicated data communications that will otherwise arise if the indirect array accesses from different iterations refer to the same array element. We need to choose the optimal region size of a *global array* to be represented by a hash table element. This will depend on the size of the *global array*, data access patterns and the number of processes and needs to be further explored. The smaller the array regions, the more small communications are generated. But if we choose a larger array region, the generated communication may include more unnecessary non-local data. Another task of the inspector is to determine which iterations access only local data, so that we may overlap non-local data communication with local data computation.

```
DO iteration=local_low, local_high
   If (this iteration contains non-local data) then
      Store the indices of non-local array elements into a hash table
      Save current iteration number in a nonlocal list
   Else
      Save current iteration number in a local list
   Endif
Enddo
Merging contiguous communications shown in the hash table
```

**Fig. 4.** An inspector pseudo-code

One optimization of the inspector used in our experiment is to merge neighboring regions into one large region in order to reduce the number of communications. The inspector loop only needs to be performed once during execution of the *gccg* program, since the indirection array remains unmodified throughout the program. Our inspector is lightweight because: 1) the location of *global arrays* is easy to compute in GA due to the simplicity of GA's data distributions; 2) the hash table approach removes enables us to identify and eliminate redundant communications; 3) all the computations of the inspector are carried out independently by each process. These factors imply that the overheads of this approach are much lower than is the case in other contexts and that it may be viable even when data access patterns change frequently, as in adaptive applications. For example, an inspector implemented using MPI is less efficient than our approach as each process has to generate communications for both sending and receiving, which rely on other processes' intervention.

The executor shown in Fig. 5 performs the computation in a parallel loop following the iteration order generated by the inspector. It prefetches non-local data via non-blocking communication, here using the non-blocking *get* operation *ga_nbget*() in GA. Simultaneously, the iterations that do not need any non-local data are executed so that they are performed concurrently with the communication. *ga_nbwait*() is used to ensure that the non-local data is available before we perform the corresponding computation.

We need to optimize the GA codes translated under this strategy. In particular, GA *get* and *put* operations created before and after each parallel construct may be redundant. If a previous *put* includes the data of a subsequent *get* operation, we may remove

this *get* operation; if the data in a *put* operation contains the content of a following *put* operation, the latter *put* operation may be eliminated. It is advantageous to move *get* and *put* operations as early as possible in order to make data available. We intend to develop array region analysis and parallel control flow analysis [3] for parallel programs in order to provide the context-sensitive array region communication information.

```
! non-local data gathering
    Call ga_nbget(….)
    DO iteration1=1, number_of_local_data
      Obtain the iteration number from the local list
      Perform the local computation
    Enddo
! wait until the non-local data is gathered
    Call ga_nbwait()
    Do iteration2=1, number_of_nonlocal_data
       Obtain the iteration number from the non-local list
       Perform computation using non-local data
    enddo
```

**Fig. 5.** An executor pseudo-code

## 3 OpenMP Extensions for Data and Work Distributions

We do not attempt to introduce data distribution extensions to OpenMP. Data distribution extensions contradict the OpenMP philosophy in two ways: first, in OpenMP the work assignment or loop schedule decides which portion of data is needed per thread and second, the most useful data distributions are those that assign data to processes elementwise, and these will require rules for argument passing at procedure boundaries and more. This would create an added burden for the user, who would have to determine data distributions in procedures that may sometimes be executed in parallel and sometimes sequentially, along with a variety of other problems. In contrast to the intended simplicity of OpenMP, existing data distribution ideas for OpenMP are borrowed from HPF and are quite complicated. For example, TEMPLATE and ALIGN are intricate concepts for non-experts. Providing equivalent C/C++ and Fortran OpenMP syntax is a goal of OpenMP; it is hard to make data distributions work for C pointers although distributing Fortran or C arrays is achievable. Finally, data distribution does not make sense for SMP systems, which are currently the main target of this API.

We rely on OpenMP static loop scheduling to decide data and work distribution for a given program. We determine a data distribution for those shared objects that will be defined as *global arrays* by examining the array regions that will be accessed by the executing threads. Data locality and load balancing are two major concerns for work distribution. Ensuring data locality may increase a load imbalance, and vice versa. Our optimization strategy gives data locality higher priority than load balancing.

We may need additional user information to minimize the number of times the inspector loop is executed. If a data access pattern in a certain code region has not

changed, it is not necessary to perform the expensive inspector calculations. Both run-time and language approaches have been proposed to optimize the inspector-executor paradigm in a given context. One of the approaches that we are experimenting with is the use of INVARIANT and END INVARIANT directives to inform compilers of the dynamic scope of an invariant data access pattern. It is noncompliant to branch into or out of the code region enclosed by INVARIANT directives. The loop bounds of parallel loops and the indirection arrays are candidate variables to be declared in INVARIANT directives, if they indeed do not change. For example, the loop bounds *NINTCI* and *NINTCF* and indirection array *LCC* are declared as INVARIANT in Fig. 6. Therefore, the communications generated by an inspector loop can be reused.

```
!$OMP INVARIANT ( NINTCI:NINTCF, LCC )
!$OMP PARALLEL
          DO I = 1, iter
           ……
!$OMP DO
           DO 4 NC=NINTCI,NINTCF
               DIREC2(NC)=BP(NC)*DIREC1(NC)
         X          - BS(NC)  * DIREC1(LCC(1,NC))
         X          - BW(NC) * DIREC1(LCC(4,NC))
         X          - BL(NC)  * DIREC1(LCC(5,NC))
         X          - BN(NC)  * DIREC1(LCC(3,NC))
         X          - BE(NC)  * DIREC1(LCC(2,NC))
         X          - BH(NC) * DIREC1(LCC(6,NC))
      4  CONTINUE
!$OMP END DO
          END DO
!$OMP END PARALLEL
!$OMP END INVARIANT
```

**Fig. 6**. A code segment of *gccg* with INVARIANT directives

In a more complex case, an application may have several data access patterns and use each of them periodically. We are exploring language extensions proposed for HPF to give each pattern a name so that the access patterns can be identified and reused, including using them to determine the required communication for different loops if these loops have the same data access pattern.

## 4 Experiments

Our initial experiments of translating regular, small OpenMP codes to the corresponding GA ones achieved encouraging results on a UH Itanium2 cluster and a NERSC IBM SP RS/6000 cluster and were reported in [10]. The UH Itanium2 cluster has twenty-four 2-way SMP nodes and a single 4-way SMP node at the University of Houston: each of the 24 nodes has two 900MHz CPUs and 4 GB memory. The Scali interconnect has a system bus bandwidth of 6.4GB/s and a memory bandwidth of 12.8GB/s. The NERSC IBM SP RS/6000 cluster is composed of 380 nodes, each of which consists of sixteen 375 MHz POWER 3+ CPUs and 16GB to 64 GB memory.

These nodes are connected to an IBM "Colony" high-speed switch via two "GX Bus Colony" network adapters. OpenMP programs can be run on a maximum of 4 processors of UH clusters and 16 processors of NERSC IBM clusters due to their SMP configuration.

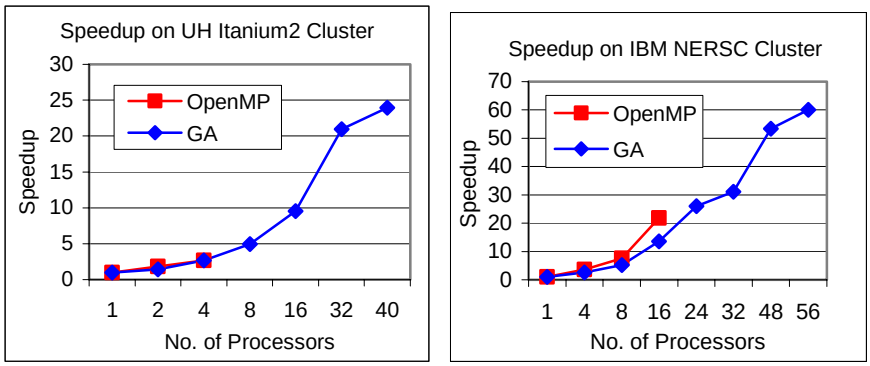


**Fig. 7**. The performance of a Jacobi OpenMP program and its corresponding GA program

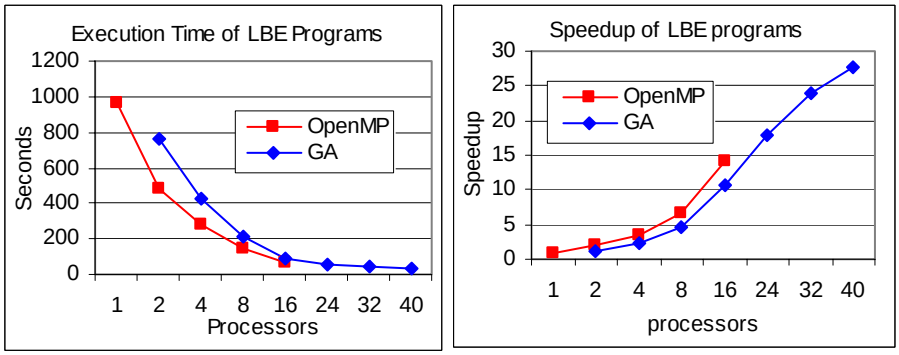


**Fig. 8.** The performance of OpenMP LBE program and the corresponding GA program

We employ implicit block data distribution for two regular OpenMP codes (Jacobi and LBE [8] ) according to OpenMP loop scheduling and our experiments show that it is feasible. Fig. 7 displays the performance of the well known Jacobi code with a 1152 by 1152 matrix on these two clusters. Both Jaocbi OpenMP program and the corresponding GA program achieved a linear speedup because of the data locality inherent in the Jacobi solver. Fig. 8 displays the performance of LBE OpenMP program and its corresponding GA program with a 1024 by 1024 matrix on NERSC IBM clusters. LBE is a computational fluid dynamics code that solves the Lattice Boltzmann equation. The numerical solver employed by this code uses a 9-point stencil. Unlike the Jacobi solver, the neighboring elements are updated at each iteration. Therefore, the performance of LBE programs is lower than that of Jacobi programs due to the writing to non-local *global arrays*. Note that our LBE program in GA was optimized to remove a large amount of synchronization; otherwise, the performance does not scale.

We experiment with FIRE benchmarks as examples of irregular codes. We use *gccg* (see also at section 2.4) to explore the efficiency of our simple data distribution, work distribution and inspector-executor strategies. Fig. 9 depicts the performance of OpenMP and corresponding GA program for *gccg* on the NERSC IBM SP cluster. The performance of OpenMP version is slightly better than the corresponding GA program within one node, but the corresponding GA program with a large input data set achieves a speedup of 26 with 64 processors in 4 nodes. The reason of performance gain is that the inspector is only calculated once and reused throughout the program, and the communication and computation are well overlapped.

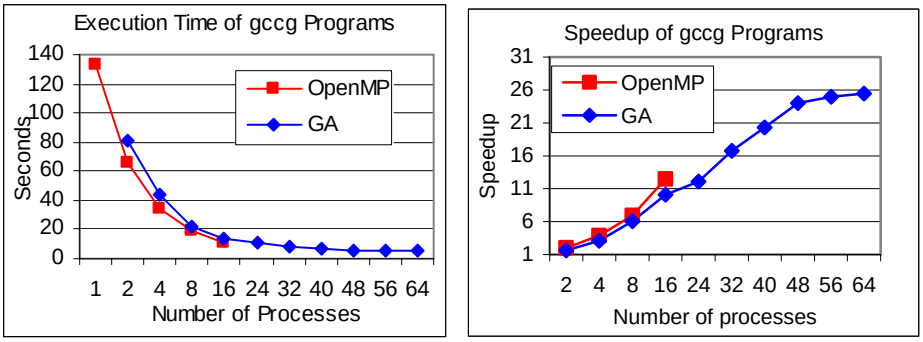


**Fig. 9**. The performance of a *gccg* OpenMP program and its corresponding GA program

## 5 Related Work

A variety of data and work distribution extensions have been proposed for OpenMP in the literature and the syntax and semantics of these differ. Both SGI [19] and Compaq [2] support DISTRIBUTE and REDISTRIBUTE directives with BLOCK, CYCLIC and * (no distribution) options for each dimension of an array, in page or element granularity. However, their syntax varies. In the Compaq extensions, ALIGN, MEMORIES and TEMPLATE directives are borrowed from HPF and further complicate OpenMP. Both SGI and Compaq also supply directives to associate computations with locations of data storage. Compaq's ON HOME directives and SGI's DATA AFFINITY directives indicate that the iterations of a parallel loop are to be assigned to threads according to the distribution of the data. SGI also provides THREAD affinity. The Portland Group Inc. has proposed data distribution extensions for OpenMP including the above and GEN_BLOCK, along with ON HOME directives for clusters of SMP systems [14]. INDIRECT directives were proposed in [11-12] for irregular applications. The idea is to create inspectors to detect data dependencies at runtime and to generate executors to remove the unnecessary synchronization; the overhead of the inspector can be amortized if a SCHEDULE reuse directive [11-12] is present.

We transparently determine a data distribution which fits into the OpenMP philosophy and search for ways to enable good performance even if this distribution is suboptimal. GA helps in this respect by providing efficient asynchronous communication.

The simplicity of the data distributions provided by GA implies that the calculation of the location of shared data is easy. In particular, we do not need to maintain a replicated or distributed *translation table* consisting of the home processors and local index of each array element, as was needed in the complex parallel partitioner approach taken in the CHAOS runtime library for distributed memory systems [4]. Our strategy for exploiting the inspector-executor paradigm also differs from the previous work, since we are able to fully parallelize the inspector loops and can frequently overlap the resulting communication with computation. Also, the proposed INVARIANT extension for irregular codes is intuitive for users and informative for the compiler implementation.

Our approach of implementing OpenMP for distributed memory systems has a number of features in common with approaches that translate OpenMP to Software DSMs [9, 18] or Software DSM plus MPI [6] for cluster execution. All of these methods need to recognize 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, whereby it is typically the latter that leads to problems. Note that it is particularly helpful to perform an SPMD 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 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 have the limitation of the shared memory spaces. An everything-shared SDSM is presented in [4] to overcome the relaxation of the coherence semantic and the limitation of the shared areas in other SDSMs. It does solve the commonly existing portability problem in SDSMs by using an OpenMP runtime 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. A remote datum can be fetched even if we do not know the owner and local index of that datum. Third, extra messages may occur in Software DSMs due to the fact that data is brought in at a page granularity. GA is able to efficiently transfer sets of contiguous or strided data, which avoids the overheads of the transfers at page granularity of Software DSMs. Besides, for the latter, the different processes have to synchronize when merging their modifications to the same page, even if those processes write to distinct locations of that page. In contrast, extra messages and synchronization are not necessary in our GA translation scheme. Our GA

approach relies on compiler analyses to obtain precise information on the array regions accessed; otherwise, conservative synchronization is inserted to protect the accesses to *global arrays*.

## 6 Conclusions and Future Work

Clusters are increasingly used as compute platforms for a growing variety of applications. It is important to extend OpenMP to clusters, since for many users a relatively simple programming paradigm is essential. However, it is still an open issue whether language features need to be added to OpenMP for cluster execution, and if so, which features are most useful. Any such decision must consider programming practice, the need to retain relative simplicity in the OpenMP API, and must take C pointers into consideration.

This paper describes a novel approach to implementing OpenMP on clusters by translating OpenMP codes to equivalent GA ones. This approach has the benefit of being relatively straightforward. We show the feasibility and efficiency of our translation. Since part of the task of this conversion is to determine suitable work and data distributions for the code and its data objects, we analyze the array access patterns in a program's parallel loops and use them to obtain a block-based data distribution for each shared array in the OpenMP program. The data distribution is thus based upon the OpenMP loop schedule. This strategy has the advantage of relative simplicity together with reasonable performance, without adding complexity to OpenMP for both SMP and non-SMP systems. Some optimizations are performed to improve the data locality of the resulting code and to otherwise reduce the cost of communications. Although load imbalance may sometimes be increased, in general the need for data locality is greater.

The inspector-executor approach is applied to enable us to handle indirect accesses to shared data in irregular OpenMP codes, since we are unable to determine the required accesses to *global arrays* at compile time. Furthermore we propose a new directive called INVARIANT to specify the dynamic scope of a program region in which a data access pattern remains invariant, so as to take advantage of the accesses calculated by an inspector loop.

In future, we plan to provide a solid implementation of our 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. The rich set of analyses and optimizations in Open64 may help us create efficient GA codes.

## 7 Acknowledgements

We are grateful to our colleagues in the DOE Programming Models project, especially Ricky Kendall who helped us understand GA and discussed the translation from OpenMP to GA with us.

## References


1. Bachler, G., Greimel, R.: “Parallel CFD in the Industrial Environment,” *Unicom Seminars*, London, 1994.
2. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C. A. and Offner, C. D.: “Extending OpenMP for NUMA machines”. *Scientific Programming*. 8(3), 2000.
3. Chakrabarti, S., Gupta, M. and Choi, J.-D.: “Global Communication Analysis and Optimization,” *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 68-78, 1996.
4. Costa, J. J., Cortes, T., Martorell, X., Ayguade, E., and Labarta, J.: “Running OpenMP Applications Efficiently on an Everything-Shared SDSM”, Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS ’04), IEEE, 2004.
5. Das, R., Uysal, M., Saltz, J., and Hwang., Y.-S.: “Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures,” *Journal of Parallel and Distributed Computing*, 22(3): 462-479, September 1994.
6. Eigenmann R. et. al.: “Is OpenMP for Grids?” *Workshop on Next-Generation Systems, Int'l Parallel and Distributed Processing Symposium (IPDPS'02)*, May, 2002.
7. Fagerström J., Faxen, Münger P., Ynnerman A. and Desplat J-C.: “High Performance Computing Development for the Next Decade, and its Implications for Molecular Modeling Applications”. http://www.enacts.org/hpcroadmap.pdf. *Daily News and Information for the Global Grid Community*, vol. 1 no. 20, October 28, 2002.
8. He, X., and Luo, L.-S.: “Theory of the Lattice Boltzmann Method: From the Boltzmann Equation to the Lattice Boltzmann Equation”. *Phys. Rev. Lett. E*, 6(56), pp. 6811, 1997.
9. Hu, Y.C., Lu, H., Cox, A. L., and Zwaenepoel, W.: “OpenMP for Networks of SMPs," *Journal of Parallel Distributed Computing*, vol. 60, pp. 1512-1530, 2000.
10. Huang, L., Chapman, B. and Kendall, R.: “OpenMP for Clusters”. *Proceedings of the Fifth European Workshop on OpenMP (EWOMP'03)*, Aachen, Germany September 22-26, 2003.
11. Hwang, Y.-S., Moon, B., Sharma, S. D., Ponnusamy, R., Das, R. and Saltz, J. H.: “Run-time and Language Support for Compiling Adaptive Irregular Problems on Distributed Memory Machines,” *Software Practice and Experience*, 25(6):597-621, June 1995.
12. Labarta J., Ayguadé E., Oliver J. and Henty D.: “New OpenMP Directives for Irregular Data Access Loops,” *2nd European Workshop on OpenMP (EWOMP'00)*, Edimburgh (UK), September 2000.
13. Liu, Z., Chapman, B. M., Weng, T.-H., Hernandez, O.: “Improving the Performance of OpenMP by Array Privatization,” *WOMPAT 2002*, pp. 244-259, 2002
14. Merlin, J.: “Distributed OpenMP: Extensions to OpenMP for SMP Clusters,” *2nd European Workshop on OpenMP (EWOMP'00)*, Edimburgh (UK), September 2000.
15. Nieplocha, J., Harrison, R. J., and Littlefield, R. J.: “Global Arrays: A non-uniform memory access programming model for high-performance computers,” *The Journal of Supercomputing*, 10, pp. 197-220, 1996
16. Open64 Compiler Tools. http://open64.sourceforge.net/
17. Saltz J., Berryman, H., Wu, J.: “Multiprocessors and Run-Time Compilation,” Concurrency: Practice and Experience, 3(6): 573-592, December 1991.
18. Sato, M., Harada, H., Hasegawa, A., and Ishikawa, Y.: "Cluster-Enabled OpenMP: An OpenMP Compiler for SCASH Software Distributed Share Memory System,” *Scientific Programming, Special Issue: OpenMP*, 9(2-3), pp. 123-130, 2001.
19. 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.
20. Top 500 Supercomputer Sites. http://www.top500.org/.