Towards a More Efficient Implementation of OpenMP for Clusters via Translation to Global Arrays*

Lei Huang, Barbara Chapman, Zhenying Liu

Department of Computer Science
University of Houston
Houston, TX, 77204, USA
`http://www.cs.uh.edu`

Keywords: OpenMP translation, Global Arrays, parallel programming languages, Distributed Memory System

Abstract

This paper discusses a novel approach to implementing OpenMP on clusters. Traditional approaches to do so rely on Software Distributed Shared Memory systems to handle shared data. We discuss these and then introduce an alternative approach that translates OpenMP to Global Arrays (GA), explaining the basic strategy. 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 the user-supplied OpenMP static loop schedules. 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. We also illustrate how to deal with some hard cases containing reshaping and strided accesses during the translation. Our experiments show promising results for the corresponding regular and irregular GA codes.

Towards a More Efficient Implementation of OpenMP for Clusters via Translation to Global Arrays*

Lei Huang, Barbara Chapman, Zhenying Liu

Abstract

This paper discusses a novel approach to implementing OpenMP on clusters. Traditional approaches to do so rely on Software Distributed Shared Memory systems to handle shared data. We discuss these and then introduce an alternative approach that translates OpenMP to Global Arrays (GA), explaining the basic strategy. 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 the user-supplied OpenMP static loop schedules. 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. We also illustrate how to deal with some hard cases containing reshaping and strided accesses during the translation. Our experiments show promising results for the corresponding regular and irregular GA codes.

Index Terms

OpenMP translation, Global Arrays, parallel programming languages, Distributed Memory System

I. INTRODUCTION

OpenMP is the de facto parallel programming standard for shared memory systems; however, it is not available for distributed memory systems including clusters, which are very widely deployed. But clusters already dominate the Top500 list [1] and the trend toward use of clusters is expected to continue according to a recent survey of hardware vendors [2]. 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 [3] and Omni/SCASH [4]. 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. [5] 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 this paper, we discuss an alternative approach that translates OpenMP programs to Global Arrays (GA) [6]. 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 remainder of this paper is organized as follows. We first describe the traditional approach of implementing OpenMP on clusters via SDSM. We then show an alternative approach that translates OpenMP to GA in Section 3. Case studies are discussed in Section 4. Section 5 describes language extensions and compiler strategies that are needed to implement OpenMP on clusters. Related work and conclusions are described in the subsequent sections.

II. TRANSLATION TO SOFTWARE DSMs

The goal of a software distributed memory system (SDSM) is to transparently implement shared memory paradigms in a distributed memory environment such as a cluster of PCs without any modifications to the source program. Some SDSMs have their own APIs; in some instances, a translator has been provided to convert shared memory paradigms such as OpenMP into appropriate calls to APIs of SDSMs. There are also a few SDSMs that were designed to implement OpenMP. The shared memory abstraction is provided via either a relaxed consistency model (e.g. TreadMarks [3] and SCASH [4]) to improve performance or a restricted sequential consistency model (e.g. NanosDSM) to increase the portability of the implementation.

Enhancements to SDSMs and preparatory compiler optimizations are employed to improve the performance of SDSM-based code. The dominant performance problems are the barriers that realize global synchronization and the need to maintain coherence of the shared memory. Most SDSMs relax coherency semantics and impose a size limitation on the shared area in order to decrease the cost of coherence maintenance. SDSMs use memory management support to detect accesses to shared memory, mostly at the granularity of pages; however, such SDSMs suffer from *false sharing* when one processor writes to a page and another processor reads or writes a different area of the same page. In this case, the writing processor will invalidate all copies of the page other than the one in its local cache, which results in a page fault when the latter processor attempts to use one. The multiple writer protocol [3], [4], [7] allows different processors to write a page at the same time and merges their modifications at the next synchronization point, thereby reducing the false sharing problem to some extent. False sharing can also be alleviated by using a finer granularity, for instance, cache-line sized granularity [8] and single-byte granularity [9]. Since a naive translation of realistic shared memory parallel programs for clusters via SDSMs cannot provide satisfactory performance [7], program optimizations prior to this translation have been proposed to obtain an efficient generated code: in particular, these target barrier elimination, intensive data privatization (replacing shared variables by threadprivate ones), page placement and data distribution.

A. TreadMarks

TreadMarks was the first system [3] that implemented OpenMP on a distributed memory system via this approach. TreadMarks has its own API for emulating shared memory. Thus the SUIF toolkit was used to build a source-to-source translator to process OpenMP directives. The OpenMP to TreadMarks translation process was relatively straightforward: OpenMP synchronization directives were replaced by TreadMarks synchronization operations; parallel regions in OpenMP were encapsulated into separate functions and translated into fork-join code. When implementing OpenMP's data environment, actual parameters passed to procedures were converted to shared data if necessary, as the translator does not perform interprocedural analysis and cannot determine whether they will be accessed from within a parallel region. Shared variables are allocated on the shared heap; private variables are allocated on TreadMark's stacks.

The TreadMarks system has emphasized work to improve memory coherence and synchronization. A lazy invalidation version of release consistency (RC) and a multiple-writer protocol were employed to reduce the amount of communication involved in implementing the shared memory abstraction. RC is a relaxed memory consistency model that categorizes shared memory accesses into acquire and release accesses, which will be realized using lock acquire and release operations.

RC allows a thread to buffer multiple writes to shared data in its local memory until a synchronization point is reached. The concepts of *twin* and *diff* were introduced in TreadMarks to reduce the cost of whole page movement. For the first write to a shared page, a *twin*, a copy of that page, is created; at barriers, a *diff* stores the modifications to the page, created by comparing the twin and the current page.

Synchronization is provided by barriers and locks. Barrier arrivals are modeled as lock releases, and barrier departures are treated as lock acquires. A centralized manager is used to control barriers: it receives a release message of a barrier arrival from each thread and broadcasts a departure message to all the threads about barrier departure after all threads have arrived the same barrier. Per-page mutex was added to allow greater concurrency in the page default handler.

A prototype compiler [5] was built to accept OpenMP and targeted TreadMarks as well as MPI library routines to enhance performance. A commercial system based upon TreadMarks is under development by Intel.

B. Omni/SCASH

For execution on clusters, the Omni OpenMP compiler translates an OpenMP C/Fortran program to a C/Fortran program with calls to a multi-threaded runtime system for SMPs or to a page-based SDSM called SCASH [4]. Omni is written in Java. When SCASH is targeted, shared data will be allocated in a shared memory area that is managed by SCASH and accessible via its primitives, since only the memory allocated via SCASH at runtime is shared among the processors. SCASH uses the concept of the *home node* of a page, which is responsible for storing the current contents of the page and which can invalidate other copies of the page; for this, *invalidate* and *update* consistency protocols are supported in SCASH. SCASH is also based on a release consistency memory model that allows multiple writers. The memory consistency is maintained at each synchronization point and only the modified parts of a page are communicated to update the home node's copy of the page. Data mapping directives in the manner of *block* and *cyclic(n)*, borrowed from HPF, and *affinity scheduling* are extended to specify how arrays are mapped to processors and how loop iterations are assigned to processors associated with the data mapping. Data mapping and affinity scheduling may enhance data locality and remove some memory consistency costs. However, the performance degrades if the data access pattern does not match the data mapping, and data remapping may be required in this case.

C. ParADE

ParADE (Parallel Application Development Environment) [10] is an OpenMP programming environment for SMP cluster systems on top of a multithreaded SDSM and message passing primitives. It provides a hybrid execution model consisting of message passing and a shared address space, attempting to overcome the poor performance of codes executed under other SDSMs by employing MPI. ParADE is based upon the Omni compiler and a Home-based Lazy Release Consistency (HLRC) protocol that avoids creating a *twin* for the modified page as all *diffs* are merged into the home page. The ParADE OpenMP translator transforms an OpenMP program into a C code with calls to the ParADE API which in turn invokes POSIX threads and MPI routines to provide the required functionality. Its developers note that an SDSM program moves much larger amounts of data between nodes than an equivalent MPI program; they also point out that the synchronization operations of an SDSM have poor performance due to the lock mechanism employed. In an attempt to provide better solutions for these two problems, the ParADE runtime system uses the message-passing primitives to avoid the conventional lock-based synchronization processes and barrier operations imposed implicitly by the OpenMP standard where possible. Message passing primitives can replace the costly locks for implementing critical sections in OpenMP, for instance, since mutual exclusion between processes is implicitly given. Collective MPI communications further contribute to reducing the number of barriers imposed by the work-sharing directives since a global synchronization is included implicitly in the collective communications. Acquiring and releasing of locks is also eliminated in the translation of SINGLE directives into ParADE routines since a global MPI broadcast suffices.

D. NanosDSM

NanosDSM, an "everything-shared" SDSM [11], was directly implemented in *NthLib*, an OpenMP run-time library designed to support a distributed memory environment. *NthLib* implements parallelism using a work descriptor that consists of a pointer to the function to be executed in parallel and its arguments. A work descriptor is stored locally and maintains consistency between threads via a message queue. A major goal of this work is to provide a portable realization of SDSM, and thus sequential consistency is offered. Spin-locks are used for all the threads to join in order to implement barriers. Several OpenMP programs, including Ocean, EP and CG (the last two are CFD

codes from the NASA Ames NAS OpenMP benchmarks), were translated by both NanosDSM and TreadMarks and subsequently executed on a distributed memory system with up to eight processors. The results show that the performance of NanosDSM programs is about the same as that achieved via TreadMarks.

E. FDSM

FDSM [9] is an OpenMP-ready SDSM system based on Omni/SCASH. FDSM analyzes the access patterns to shared memory during the first iteration of a loop and obtains the communication set in order to reduce the memory coherence overhead. A single-byte granularity is employed for write operations in order to decide which variables were modified; however, page granularity is used to detect the read operations.

FDSM calculates the communication set for each OpenMP parallel loop and its parent loop. If a variable is written in one block and another processor reads it in another block, communication is required to update the variable. Function calls are inserted into the source code in order to start and complete acquiring the data associated with an access pattern. After obtaining the communication set, at the memory barrier point, FDSM synchronizes to ensure that no process accesses the shared memory, copies the data to remote memory, and synchronizes again to ensure data updating. As with other SDSMs, the page granularity at which this is handled prevents precise analysis of the data read, so that false sharing in particular leads to unnecessary communication. It is thus likely to communicate larger amounts of data than a corresponding manually-developed MPI. Experiments reported using FDSM to execute the CG code from the NAS OpenMP benchmarks on an Intel Pentium III cluster showed linear scalability on up to 8 processors.

III. TRANSLATION TO GLOBAL ARRAYS

In this section, we introduce an alternative approach to implementing OpenMP on distributed memory systems, that is, our translation from OpenMP to GA [6]. The main benefit of this approach, and the reason we explore it, is that it makes all communications explicit in the code, thereby allowing us to explicitly control and optimize data communications, but at the same time it provides a shared memory abstraction that makes most parts of the translation straightforward. We first describe the basic translation, and then consider some advanced problems.

A. A Basic Translation to GA

GA [12] 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.

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. 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 [6] strategy follows OpenMP semantics. 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. 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*.

Our basic translation strategy assigns iterations of OpenMP parallel loops (OMP DO) 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.

```
1    !$OMP PARALLEL SHARED (a)
2        do k=1, MAX
3    !$OMP DO
4        do j = 1, SIZE_Y
5          do i = 2, SIZE_X−1
6            a(i,j) = a(i+1,j) + ...
7          enddo
8        enddo
9    !$OMP END DO
10       enddo
11   !$OMP END PARALLEL
```

(a) Original OpenMP program

```
1     call  MPI_INIT()
2     call  ga_initialize()
3     status=ga_create(MT_DBL, SIZE_X , SIZE_Y , 'A' ,
4             SIZE_X, SIZE_Y/ nproc , g_a )
5     Do k = 1 , MAX
6   !compute new low bound and upper bound for each processs
7        (new_low , new_upper) = ...
8   !compute the array read region for each thread
9        (ilow , ihi , jlow , jhi)=  (1, SIZE_X, ...)
10       call ga_get( g_a , ilow , ihi , jlow , jhi , a, ld )
11       call ga_sync( )
12       num_thread = compute_thread_number_needed()
13       do j = new_low , new_upper
14         do i = 2 , SIZE_X − 1
15             a (i , j) = a(i+1, j) + ...
16         enddo
17       enddo
18  !compute array write region and put the modified data back to GA
19       (ilow , ihi , jlow , jhi) = (2 , SIZE_X−1, ...)
20       call ga_put(g_a , ilow , ihi , jlow , jhi , a(2,new_low) , ld )
21       call ga_sync( )
22   enddo
```
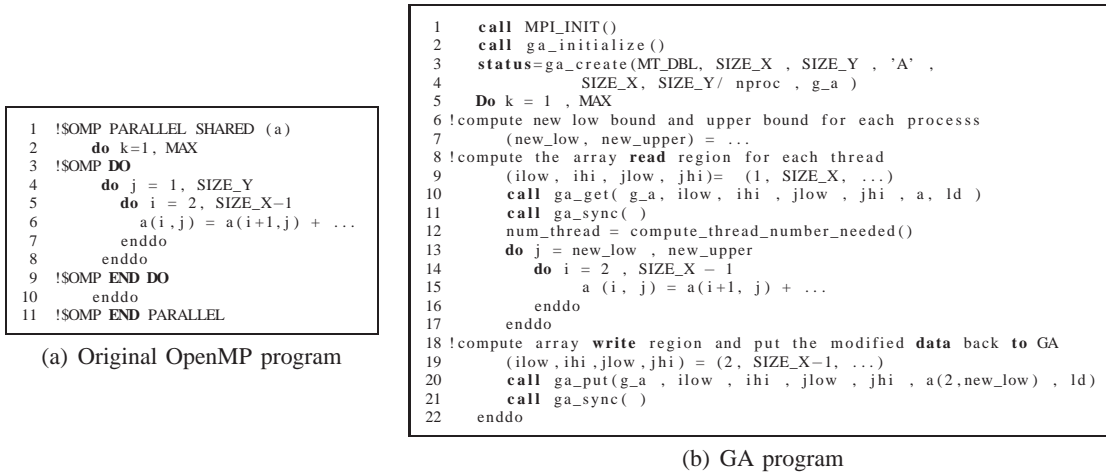
(b) GA program

Fig. 1.   An OpenMP program and corresponding GA+OpenMP program

We show an example of an OpenMP program in Fig. 1(a) and the corresponding GA program, obtained by applying the basic translation strategy, 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.

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 run-time. 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.

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. 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. The OpenMP SECTION, SINGLE and MASTER directives can be translated into GA using different strategies and we will explain them in the next section. 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.

We may perform global privatization and transform OpenMP into the so-called SPMD style [13] before the translation occurs 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.

B. Implementing Sequential Regions

All strategies for implementing OpenMP on clusters have a problem with sequential regions, since they may require access to shared data and their results may be used in all threads. In our approach, 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 written 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 *global 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.

C. 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 paradigm 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 implement 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(as specified in HPF [14], this generalized block distribution permits the assignment

of contiguous segments of uneven length to processors); otherwise, a BLOCK distribution is employed. 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 suitable programming tools could collaborate with the user to improve this translation in many cases. One way to enhance this approach is to perform data distribution based on the most time-consuming parallel loops. 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. Alternatively, a static estimation or profile results, even if based on a partial execution, may be exploited. 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 although it is possible to implement all forms of OpenMP loop schedule including OpenMP static, dynamic and guided loop scheduling, our current approach does not handle all of these well. OpenMP static loop scheduling distributes iterations evenly. When the iterations of a parallel loop have different amount of work, dynamic and guided loop scheduling may be used to balance the work-load. We can realize the work assignment in GA that corresponds to a dynamic or guided loop schedule; 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.

IV. CASE STUDIES

Our initial experiments involved translating regular, small OpenMP codes to the corresponding GA ones. They achieved encouraging results on a UH Itanium2 cluster and a NERSC IBM SP RS/6000 cluster and were reported in [6]. 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.

Our first experiments translated two relatively simple regular OpenMP codes (Jacobi and LBE [15]) according to the strategy described above. The loop schedules consistently corresponded to block data distributions for these codes. Fig. 2 displays the performance of the well known Jacobi code with a 1152 by 1152 matrix on these two clusters. Both the OpenMP Jaocbi program and the corresponding GA program achieved a linear speedup because of the data locality inherent in the Jacobi solver. Fig. 3 displays the performance of the LBE OpenMP program and its corresponding GA program with a 1024 by 1024 matrix on the NERSC IBM cluster. 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 more extensive writing to non-local elements of global arrays. Note that our LBE program in GA was manually optimized to remove a large amount of
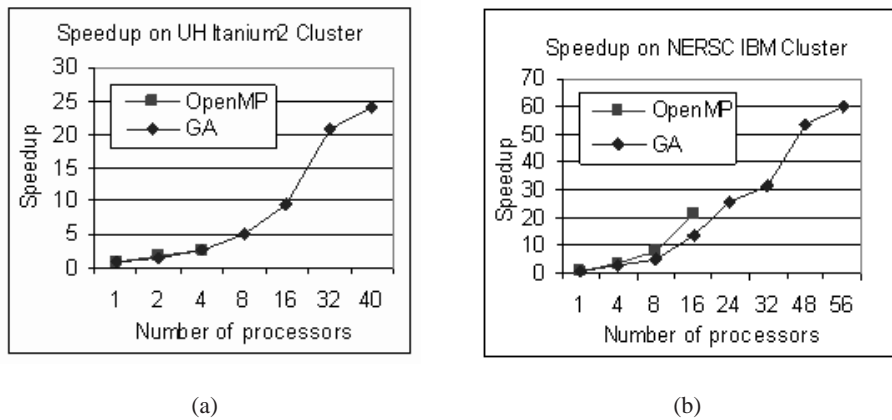
Fig. 2.   The performance of a Jacobi OpenMP program and its corresponding GA program
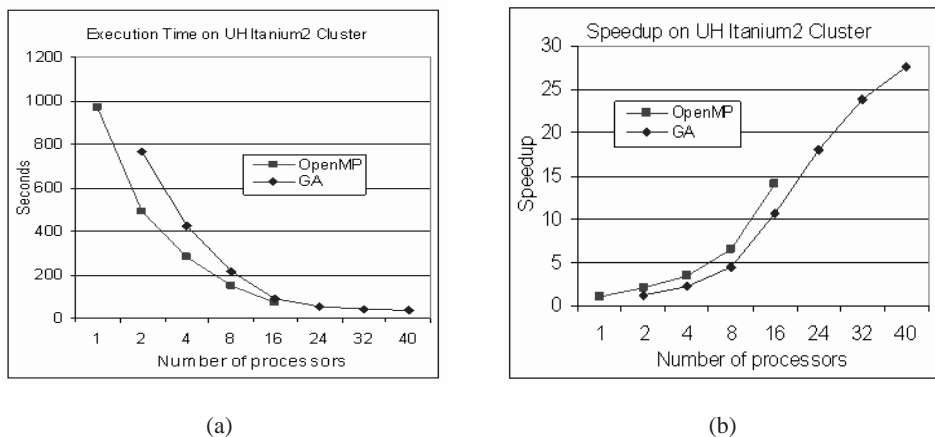

Fig. 3.   The performance of a LBE OpenMP program and its corresponding GA program

synchronization; otherwise, the performance does not scale. The most important optimization we have implemented in LBE GA program was the communication aggregation, which combines small messages into a big message and reduces the communication overheads. We also used the non-block *get/put* operations to replace the block *get/put* operations and moves these operations as early/late as possible in order to overlap some of the communication and computation. We also removed some redundant communication operations.

A. FIRE Benchmarks

We studied one of the programs, *gccg*, in the FIRE Benchmarks [16] in order to understand how to translate codes containing irregular data accesses to GA. FIRE is a fully interactive fluid dynamics package for computing compressible and incompressible turbulent flow, and its major data structures and access patterns are captured in the benchmark suite. *Gccg* is a parallelizable solver in the FIRE package that uses orthomin and diagonal scaling.

We show the process of translating the OpenMP *gccg* program into the corresponding GA program. Fig. 4 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 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 *global arrays* according to the work distribution in the two parallel loops in Fig. 4. 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

```
1  !$OMP  PARALLEL
2            DO I = 1 , iter
3  !$OMP  DO
4            DO 10  NC=NINTCI ,NINTCF
5               DIREC1 (NC)=DIREC1 (NC)+RESVEC(NC)*CGUP(NC)
6         10  CONTINUE
7  !$OMP  END DO
8  !$OMP  DO
9             DO 4  NC=NINTCI ,NINTCF
10               DIREC2 (NC)=BP(NC)*DIREC1 (NC)
11        X          − BS(NC)   *  DIREC1 (LCC( 1 ,NC))
12        X          − BW(NC)  *  DIREC1 (LCC( 4 ,NC))
13        X          − BL(NC)   *  DIREC1 (LCC( 5 ,NC))
14        X          − BN(NC)   *  DIREC1 (LCC( 3 ,NC))
15        X          − BE(NC)   *  DIREC1 (LCC( 2 ,NC))
16        X          − BH(NC)  *  DIREC1 (LCC( 6 ,NC))
17        4   CONTINUE
18 !$OMP  END DO
19            END DO
20 !$OMP  END PARALLEL
```

Fig. 4.   An OpenMP code segment in gccg with irregular data accesses

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 statically determinable block-based data distribution strategy *DIREC1* in the second loop. The 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 the first process and the last process respectively for contiguity. Therefore, it is not an even distribution and a GEN_BLOCK distribution is employed as shown in Fig. 5, assuming four processors are targeted.
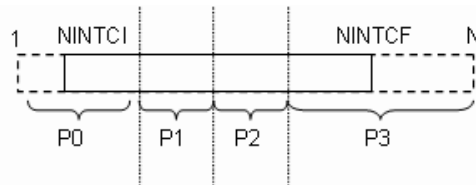

Fig. 5.   GEN_BLOCK distribution for array DIREC1

As before, we perform work distribution according to the OpenMP loop scheduling in Fig. 4. Fortunately, we do not need to insert any communications for the first loop in Fig. 4 since all accesses to shared data are statically known to be local. But in the second loop of Fig. 4, some data accesses are based upon subscript expressions containing array references (i.e. are indirect) and thus cannot be further analyzed at compile time. Therefore we cannot determine the required communications and generate efficient communication calls based upon static compiler analysis. A suitably adapted inspector-executor strategy [17] is employed to handle such situations.

The inspector-executor approach [18] generates an extra inspector loop preceding the actual computational loop. Our inspector is a parallel loop as shown in Fig. 6. 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 or nearby elements. We need to choose the optimal size of a *global array* region to be represented by a hash table element. This will depend on the size of the *global array*, the 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 large 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.

One important optimization of our inspector approach is to merge neighboring regions into one larger region in

```
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
Merge contiguous communications based on hash table entries
```

Fig. 6.   Inspector pseudo-code

order to reduce the number of communications. We must also attempt to statically determine whether the subscript expressions in the indirect references are modified at run time (we return to this further below). The inspector loop only needs to be performed once during execution of the *gccg* program, since here the indirection array remains unmodified throughout the program. Our inspector is lightweight because: 1) the location of individual elements of *global arrays* is easy to compute in GA due to the simplicity of GA's data distributions; 2) the hash table approach 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 over time, as occurs 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. 7 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.

```
! gather non-local data
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. 7.   Executor pseudo-code

There is further potential for optimizing the GA codes translated under this strategy. In particular, GA *get* and *put* operations created before and after each parallel construct may be redundant. For example, If a previous *put* includes the data of a subsequent get operation, we may remove this *get* operation; Two adjacent *put* operations may be merged into one operation if the data in the first operation are not accessed by other processes. It is advantageous to move *get* operations as early as possible in order to enable prefetching of data and move *put* operations as late as possible in order to merge these operations and reduce the communication overhead. In order to automatically apply these optimizations, we are working on improvements to our array region analysis and parallel control flow analysis [19] in order to provide the context-sensitive array region communication information required.

Fig. 8 depicts the performance of the OpenMP and corresponding GA programs for gccg on the NERSC IBM SP cluster. The performance of the OpenMP version is slightly better than the GA code within one node, but

with a large input data set the GA program achieves a speedup of 26 with 64 processors in 4 nodes, which is quite reasonable. One reason for this performance gain is that the inspector is only calculated once and is then reused throughout the program's execution. We also manually implemented the aforementioned optimizations for the communications and overlap the communication and computation as much as possible.
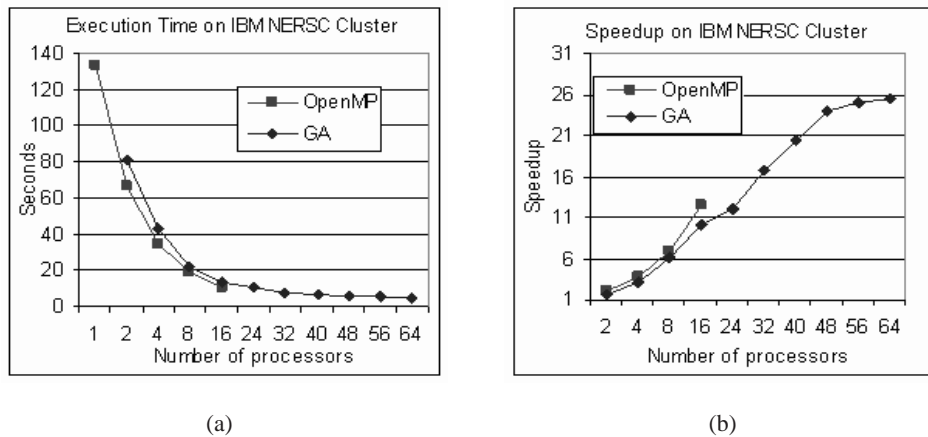

(a)   (b)

Fig. 8.   The performance of OpenMP *gccg* code and corresponding GA program

B. UMT98

UMT98 [20] is a 3D neutral particle transport code for unstructured meshes, which solves the first-order form of the steady-state neutral-particle Boltzmann transport equation. UMT98 is parallelized for SMP systems using OpenMP. It contains 44 functions and 1 parallel loop. The parallel loop sweeps over each discrete ordinate. Computations in each of the three dimensions are independent for a large class of problems. Therefore good scalability of this parallel loop is achieved. The parallel loop as shown in Fig. 9 contains 6 procedure calls that contribute the vast majority of the computational work. There are 64 shared variables inside the parallel region and most of them are arrays. Fig. 10 shows a code fragment from the routine *snxyzref*, which is one of the 6 subroutines inside the main parallel loop. The remaining subroutines called outside the parallel loop mainly initialize data.

```
1       real*8  ... , QC(npart , ncornr , n_cpu) , ...
2       ...
3 !$OMP PARALLEL DO PRIVATE(m, itet , i ,mm, mpsi ,mstride ,thnum , setbc )  ...
4     do 100 mm=1,ndir
5         m       = morder(mm)
6         thnum = mthread(mm)
7         call snqq ( ... , QC(1 ,1 , thnum ) , ...)
8         call snxyzref ( ... ,m, mtmp(1 , thnum ), ... , tmp(1 , thnum ) ,PSIB , ABDYM(1 , thnum ), A_bdy)
9         call snxyzbc ( ... , abdym(1 , thnum ) , psib , PSIFEP(1 ,1 , thnum ))
10        call snswp3d ( ... , TPSIC(1 ,1 , thnum ) ,PSIC(1 , mpsi ) , ...)
11        call snxyzbc ( ... , abdym(1 , thnum ) , PSIB , psifep (1 ,1 , thnum ))
12        call snmmnt ( ... , quadwt(m) ,ynm, psic (1 , mpsi ) ,TPHIC(1 ,1 , thnum ))
13        ...
14    enddo
```

Fig. 9.   Computation in main parallel loop in UMT98

As before, we perform the work distribution for UMT98 according to the OpenMP static loop schedule. We modify the loop bounds and distribute the loop iteration evenly into each participating process. Since UMT98 is an irregular data access application: the data access is not directly dependent on a loop index, it is very complicated or even impossible to figure out the best data distribution at the compile time. We choose the simple block data distribution for most of arrays in UMT98.

In this code, the developers have passed arrays to subroutines in such a way that the formal argument has a different number of dimensions than the original array has. For example, the 3-dimensional shared array *QC* in UMT98 that occurs in the code fragment in Fig. 9 is subsequently referenced via a 2-dimensional formal argument. This array reshaping is typical of many Fortran applications, and when shared data is referenced in this way the task of distributing data becomes difficult. In this case, *QC(1,1,thnum)* is passed to subroutine *snqq* as a formal parameter,

where *thnum* represents the thread ID. In the subroutine *snqq*, QC is declared as a 2-dimensional *QC(npart, ncornr)*. Since each process accesses a different region of array *QC*, given the distinct values in the third dimension, we may distribute this array in the third dimension so that each process accesses its local part of the array only. Hence, we are able to replace the declaration of the distributed array QC by a private array *QC(npart, ncornr)* for each thread. Several other arrays, such as *mtmp, tmp, DOT* in subroutine *snxyzref* (see Fig. 10), can be dealt with in the same manner. However, this is a significant modification that impacts all references to the original array in the source code. It goes beyond the techniques currently available to us and may best be addressed via some combination of user input, compiler feedback and manual modification, possibly within the framework of a tool.

```
1 Subroutine snxyzref(...,m,mtmp,...,tmp,psib,dot,a_bdy)
2    integer ...,m,mtmp(ndir), ...
3    real*8 ...,tmp(ndir),psib(nbelem,ndir*npart),dot(nbelem),a_bdy(ndim,nbelem)
4    integer mref,... ! local variables
5    ......
6    call snmref(nout,ndim,ndir,m,mref,cosrat,deltax,quadwt,omega,tmp,mtmp)
7    do ip=1,npart
8       moff = (ip-1)*ndir
9       do ix=ix1(ibs),ix2(ibs)
10         ib = lcx(ix)
11         psib(ib,moff+m)=cosrat*psib(ib,moff+mref) !this is psib(lcx(ix),(ip-1)*ndir +mref)
12      enddo
13   enddo
14   ...
15   return
16 end
```

(a) Subroutine *snxyzref*

```
1  Subroutine snxyzref(...,m,mtmp,...,tmp,psib,dot,a_bdy)
2     integer ...,m,mtmp(ndir), ...
3     real*8 ...,tmp(ndir),psib(nbelem,npart,ndir),dot(nbelem),a_bdy(ndim,nbelem)
4     integer mref,... !local variables
5     ......
6     call snmref(nout,ndim,ndir,m,mref,cosrat,deltax,quadwt,omega,tmp,mtmp)
7     call ga_get(g_psib,(1,1, mref),(nbelem,npart,mref),psib,ld)
8     do ip=1,npart
9        do ix=ix1(ibs),ix2(ibs)
10          ib = lcx(ix)
11          psib(ib, ip, m) = cosrat*psib(ib, ip, mref)
12       !m is calculated from loop index mm and has distinct values for each process
13          enddo
14    enddo
15    call ga_put(g_psib,(1,1, m), (nbelem,npart,m),psib,ld)
16    ...
17    return
18 end
```

(b) The corresponding GA program

Fig. 10.   Subroutine *snxyzref* called inside the only parallel loop in UMT98 and the corresponding GA program

Another hard case in practice is finding efficient ways to deal with strided accesses to an array. Although the GA implementation works to reduce the cost of strided accesses, these may nevertheless incur high communication costs. It may sometimes be possible to change the layout of the array before its distribution in order to enable contiguous accesses. To see the problems involved in practice, we consider references to the shared array *psib* in UMT98. This is declared as *psib(nbelem,ndir*npart)* in subroutine *snxyzref* (in Fig. 10(a)). The write access to this array on line 11 of the code is both indirect and strided. Since GA requires us to choose a block distribution, we will have many non-local indirect or strided accesses no matter which dimension we distribute *psib* in. In order to enable local accesses and contiguous non-local data communication, we may take the following two actions. First, we split the second dimension of *psib* and thereby obtain a 3-dimensional array *psib(nbelem,ndir,npart)* as shown in the revised *snxyzref* (Fig. 10(b)). Fig. 11 illustrates how accesses will be made to the array *psib* before and after this *array reshaping*. Second, we distribute array *psib* in the third dimension by block. To understand the impact, we must note that subroutine *snxyzref* is invoked only from within the parallel region. Within it, each thread has a distinct value for the dummy variable *m* which is the index variable of the third dimension of the array; hence, each updates a different set of elements of *psib*. It is therefore appropriate to follow the original OpenMP work assignment when distributing the reshaped array. Distributing *psib* in the newly added third dimension allows us to write to local elements and group the communications needed for obtaining non-local elements. Fig. 11 shows the different access pattern that results for an individual thread. After reshaping and data distribution in the third dimension, these data accesses are all local to a process.

We replicate the scalar variables and perform any work needed to ensure that they keep the same values in
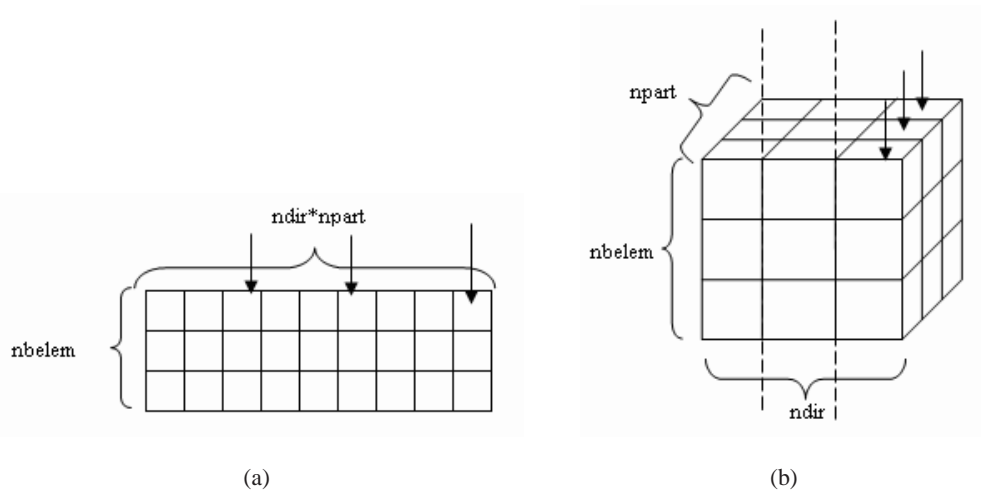
Fig. 11. Strided accesses to array psib before (a) and after (b) data reshaping and distribution

each process. We also replicate the small read-only arrays inside the parallel loop to reduce the communication overheads.

The examples above illustrate some of the challenges involved in trying to deal with real-world code. Unfortunately, they are too complex to be automatically dealt with at present and in some cases, good compile-time translations are prevented by a lack of information. For example, the aforementioned three-dimensional array *QC* is distributed in the third dimension in the original subroutine for the best data locality. However, the value of the index variable *thnum* that is used to access the third dimension of *QC* is obtained from another array, *mthread*. This complicates the compiler analysis needed to determine the data distribution of array *QC*. We believe that appropriate extensions to an interactive tool compiler-based tool, such as Dragon [21], is the most useful way to help determine and realize a translation to GA in practice when such programming practices are encountered. A compiler may help to identify some difficult cases. Informal language extensions or user input in an interactive system might provide the information needed to restructure and possibly even privatize arrays.

We also experimented with the OpenMP and GA versions of UMT98 on a SUN cluster at UH and a NERSC IBM cluster. The SUN cluster has four 4-way SUN Enterprise 420 connected by Ethernet. OpenMP is only available within an SMP node of each platform: one node of the Sun cluster contains 4 processors and one node of IBM NERSC cluster has 16 processors. Fig. 12 shows the speedup of OpenMP UMP98 and the corresponding GA code, which is based upon a manual implementation of the transformations described above. The speedup of the UMT98 GA program is linear and scales on both clusters due to the independence of computations once these extensive modifications and more suitable data distribution have been exploited.
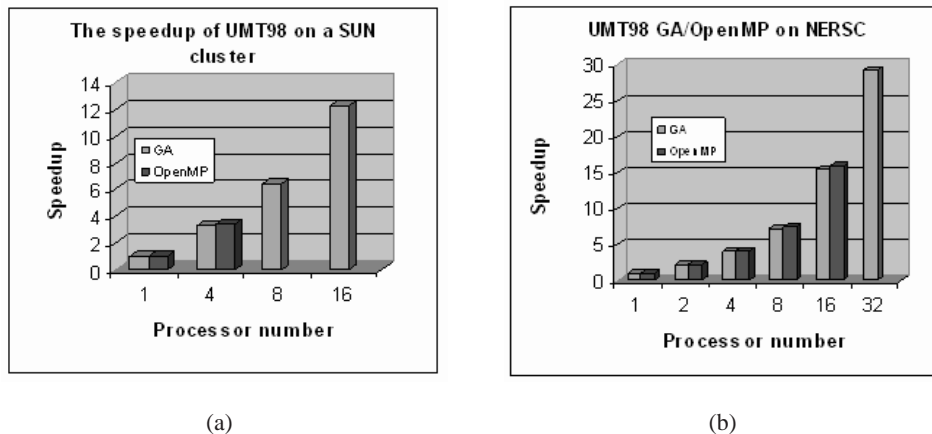

Fig. 12. Speedup of UMT98 OpenMP and corresponding GA program in a SUN cluster of UH (a) and NERSC IBM Cluster (b)

V. OpenMP Extensions for Data and Work Distributions

We do not attempt to introduce data distribution extensions to OpenMP as part of the language proper. 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 element-wise, and these 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, and adding them to the language would require their availability in all compilers.

However, informal user information could be employed to help select the data distribution, especially in difficult cases such as the last example discussed above or where different program regions imply different distributions. Our basic strategy is to 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 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 runtime 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. 13. Therefore, the communications generated by an inspector loop can be reused.

```
 1  !$OMP  INVARIANT  (  NINTCI:NINTCF,  LCC  )
 2  !$OMP  PARALLEL
 3              DO  I  =  1 ,  iter
 4                  . . .
 5  !$OMP  DO
 6               DO  4  NC=NINTCI,NINTCF
 7                   DIREC2(NC)=BP(NC)*DIREC1(NC)
 8          X              −  BS(NC)   *  DIREC1(LCC(1 ,NC))
 9          X              −  BW(NC)  *  DIREC1(LCC(4 ,NC))
10          X              −  BL(NC)  *  DIREC1(LCC(5 ,NC))
11          X              −  BN(NC)   *  DIREC1(LCC(3 ,NC))
12          X              −  BE(NC)   *  DIREC1(LCC(2 ,NC))
13          X              −  BH(NC)  *  DIREC1(LCC(6 ,NC))
14          4   CONTINUE
15  !$OMP  END DO
16               END  DO
17  !$OMP  END  PARALLEL
18  !$OMP  END  INVARIANT
```

Fig. 13.   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.

VI. Related Work

Given the importance of clusters, and the lack of a portable high-level programming model for them, it is not surprising that a number of vendors as well as researchers have considered the relevance of OpenMP to them, and a variety of data and work distribution extensions have been proposed for OpenMP in the literature to help it target such platforms as well as large-scale shared memory machines. Unfortunately, the syntax and semantics of these differ. Both SGI [22] and Compaq [23] introduced 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 the location of data. 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 specified data. SGI also provides for a direct 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 [24]. INDIRECT directives were proposed in [25] for irregular applications. The idea is to create inspector code to detect data dependences at runtime and to generate executor code to remove the unnecessary synchronization; the overhead of the inspector can be amortized if a SCHEDULE reuse directive [25] is present. In the HPF/JA language specification [26], an INDEX_REUSE directive is proposed to instruct the compiler to reuse the communication schedule for an array irregularly accessed in a loop.

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, for example, in the CHAOS runtime library for distributed memory systems [27]. Our strategy for exploiting the inspector-executor paradigm also differs from other 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 to implementing OpenMP for distributed memory systems has a number of features in common with approaches that translate OpenMP to Software DSMs [3], [4] or Software DSM plus MPI [5] 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, whereby 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 [11] 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. A remote datum can be fetched without interfering with the process that owns the datum. Third, extra

messages may be incurred when using SDSMs due to the fact that data is fetched at a page granularity. Besides, 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. GA is able to efficiently transfer sets of contiguous or strided data, which avoids the overheads of page-sized transfers. Since page sizes are growing on recent architectures, the difference may be substantial. Moreover, 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 and this is a major focus of our on-going compiler implementation; otherwise, conservative synchronization is inserted to protect the accesses to *global arrays*.

VII. 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 follow OpenMP loop schedule semantics to distribute the data. 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 show the feasibility and efficiency of our translation via experiments on regular and irregular codes. For regular applications, automatic translation of OpenMP into GA is attainable. However, codes employing techniques such as array reshaping across procedure boundaries will remain problematic. Extra effort is required for irregular codes since it is not easy for the compiler to figure out the data access patterns at compile time. The inspector-executor approach enables us to handle indirect accesses to shared data. 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.

We are working on providing a solid implementation of our translation from OpenMP to GA in the Open64 compiler [28], 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. We have also built Dragon, a graphical interactive program analysis tool that extracts information from Open64 and also is able to gather additional information. We may be able to use Dragon features to further explore combinations of user input, compiler feedback and manual modification that might make it easier to deal with some of the hard problems that are encountered in real-world application codes. Further, we will continue to compare the benefits and drawbacks of the different approaches to implementing OpenMP on clusters, and will consider whether our strategy might be fruitfully combined with other emerging techniques in order to maximize the benefits, for example exploiting existing support for shared pointers in SDSMs.

VIII. Acknowledgments

References

[1] "Top 500 supercomputers sites." [Online]. Available: {http://www.top500.org}

[2] J. Fagerstróm, A. Faxen, Múnger P.and Ynnerman, and J.-C. e. a. Desplat, "High Performance Computing Development for the Next Decade, and its Implications for Molecular Modeling Applications," October 28, 2002. [Online]. Available: http://www.enacts.org/hpcroadmap.pdf

[3] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel, "OpenMP for Networks of SMPs," *Journal of Parallel Distributed Computing*, vol. 60, pp. 1512–1530, 2000.

[4] M. Sato, H. Harada, A. Hasegawa, and Y. Ishikawa, "Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system," *Scientific Programming, Special Issue: OpenMP*, vol. 9, no. 2–3, pp. 123–130, Spring–Summer 2001.

[5] R. Eigenmann, J. Hoeflinger, R. H. Kuhn, D. Padua, A. Basumallik, S.-J. Min, and J. Zhu, "Is OpenMP for Grids?" in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '02)*, Fort Lauderdale, 2002.

[6] L. Huang, B. Chapman, and R. Kendall, "OpenMP for Clusters," in *the Fifth European Workshop on OpenMP, EWOMP'03*, Aachen, Germany, 2003.

[7] A. Basumallik, S.-J. Min, and R. Eigenmann, "Towards openmp execution on software distributed shared memory systems," in *Proceedings of the 4th International Symposium on High Performance Computing*. Springer-Verlag, 2002, pp. 457–468.

[8] Z. Radović and E. Hagersten, "Removing the Overhead from Software-Based Shared Memory," in *Proceedings of the ACM/IEEE Supercomputing 2001(SC2001) Conference*, Denver, Colorado, 2001.

[9] H. Matsuba and Y. Ishikawa, "OpenMP on the FDSM Software Distributed Shared Memory," in *the Fifth European Workshop on OpenMP, EWOMP'03*, Aachen, Germany, September, 2003.

[10] Y.-S. Kee, J.-S. Kim, and S. Ha, "ParADE: An OpenMP Programming Environment for SMP Cluster Systems," in *Proceedings of the ACM/IEEE Supercomputing 2003(SC2003) Conference*, Phoenix, Arizona, November 15 - 21, 2003.

[11] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta, "Running OpenMP Applications Efficiently on an Everything-Shared SDSM," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*. IEEE, 2004.

[12] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers," *Journal of Supercomputing*, vol. 10, pp. 169–189, 1996.

[13] Z. Liu, B. Chapman, T.-H. Weng, and O. Hernandez, "Improving the Performance of OpenMP by Array Privatization," in *Workshop on OpenMP Applications and Tools (WOMPAT 2002)*, Fairbanks, Alaska, 2002.

[14] "High performance fortran language specification. version 2.0." Rice University, Houston, TX, Tech. Rep., January 1997. [Online]. Available: http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20/index.html

[15] X. He and L.-S. Luo, "Theory of the lattice boltzmann method: From the boltzmann equation to the lat-tice boltzmann equation," *Phys. Rev. Lett. E*, vol. 6, no. 56, p. 6811, 1997.

[16] G. Bachler and R. Greimel, "Parallel CFD in the Industrial Environment," in *Unicom Seminars*, London, 1994.

[17] Y.-S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz, "Run-time and language support for compiling adaptive irregular problems on distributed memory machines," *Software Practice and Experience*, vol. 25, no. 6, pp. 597–621, June 1995.

[18] J. Saltz, H. Berryman, and J. Wu, "Multiprocessors and Run-Time Compilation," *Concurrency: Practice and Experience*, vol. 3, no. 6, pp. 573–592, December 1991.

[19] S. Chakrabarti, M. Gupta, and J.-D. Choi, "Global communication analysis and optimization," in *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*. ACM Press, 1996, pp. 68–78.

[20] "Umt98 unstructured mesh transport." [Online]. Available: {http://www.llnl.gov/asci/applications/UMT98README.html}

[21] O. Hernandez, C. Liao, and B. Chapman, "Dragon: A Static and Dynamic Tool for OpenMP," in *Proceedings of Workshop on OpenMP Applications and Tools (WOMPAT 2004)*, Houston, TX, 2004.

[22] *MIPSpro 7 FORTRAN 90 Commands and Directives Reference Manual*, Silicon Graphics Inc., accessed 2002. [Online]. Available: http://techpubs.sgi.com

[23] J. Bircsak, P. Craig, R. Crowell, *et al.*, "Extending OpenMP for NUMA Machines," *Scientific Programming*, vol. 8, no. 3, pp. 163–181, 2000.

[24] J. Merlin, "Distributed OpenMP: Extensions to OpenMP for SMP Clusters," in *2nd European Workshop on OpenMP (EWOMP'00)*, Edinburgh (UK), September, 2000.

[25] J. Labarta, E. Ayguadé, J. Oliver, and D. Henty, "New OpenMP Directives for Irregular Data Access Loops," in *the Second European Workshop on OpenMP*, Edinburgh, Scotland, 2000.

[26] J. A. for High Performance Fortran), "HPF/JA Language Specification, English Version 1.0." Nov. 11, 1999. [Online]. Available: http://www.hpfpc.org/jahpf/spec/hpfja-v10-eng.pdf

[27] R. Das, M. Uysal, J. Saltz, and Y.-S. S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 462–478, 1994.

[28] "The Open64 Compiler." [Online]. Available: {http://open64.sourceforge.net/}