Efficient Translation of OpenMP to Distributed Memory

L. Huang[1], B. Chapman[1], Z. Liu[1], and R. Kendall[2]

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

Abstract. The shared memory paradigm provides many benefits to the parallel programmer, particular with respect to applications that are hard to parallelize. Unfortunately, there are currently no efficient implementations of OpenMP for distributed memory platforms and this greatly diminishes its usefulness for real world parallel application development. In this paper we introduce a basic strategy for implementing OpenMP on distributed memory systems via a translation to Global Arrays. Global Arrays is a library of routines that provides many of the same features as OpenMP yet targets distributed memory platforms. Since it enables a reasonable translation strategy and also allows precise control over the movement of data within the resulting code, we believe it has the potential to provide higher levels of performance than the traditional translation of OpenMP to distributed memory via software distributed shared memory.

1 Introduction

Recently, programmer productivity has been emphasized for parallel and distributed computing. However, few robust high-level language models have been developed for parallel programming. The difficulty of designing a language that offers expressivity, portability, ease of use and high performance has inevitably led to many failures. The distributed memory paradigm with explicit message passing remains the de facto programming standard, mainly because it provides good scalability for regular applications and it addresses the price/performance driven evolution of the HPC market toward clustered architectures. This scalability comes at a high programming cost. The shared memory paradigm has a low entry cost and can be much more flexible in its ability to respond to dynamically changing characteristics of an application. OpenMP [1] is a popular parallel programming interface for medium scale high performance applications on shared memory platforms. Strong points are its APIs for Fortran, C and C++, the ease with which its directives can be inserted into a code, its ability to support incremental parallelization, features for dynamically setting the numbers of threads and scheduling strategies, and strong vendor support. This is offset by its lack of support for distributed memory.

There have been a variety of attempts to implement OpenMP on clusters, most of which are based upon a translation of OpenMP to a software DSM (Distributed Shared Memory) system which is then responsible for managing data declared to be shared [2, 8]. Such solutions tend to be inefficient, as the software DSM will perform

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 408–413, 2004.

expensive amounts of data transfer at each (explicit or implicit) barrier in the program, and are particularly problematic for codes that are hard to parallelize, such as unstructured computations. In this paper, we propose an alternative strategy for translating OpenMP code to execute on clusters. We believe that our strategy will be more efficient and of specific benefit for irregular (unstructured) computations.

The paper is organized as follows. In the next section, we introduce Global Arrays, a library for parallel programming upon which our translation strategy is based, and explain why we believe that this can help us implement OpenMP efficiently on clusters. The remainder of the paper discusses our translation strategy, summarizes related work and our future plans.

2 Global Arrays as a Basis for Translating OpenMP

Global Arrays (GA) [7] is a collection of library routines that was designed to simplify the programming methodology on distributed memory systems. GA has been available in the public domain since 1994 and has since been utilized to create parallel versions of many major scientific codes for distributed memory machines. It realizes a portable interface via which processes in an SPMD-style parallel program do not need the explicit cooperation of other processes. In contrast to other popular approaches, it does so by providing a library of routines that enable the user to specify and manage access to *shared* data structures in a program. Compared with MPI programming, GA thus simplifies parallel programming by providing users with a conceptual layer of virtual shared memory for distributed memory systems. Programmers can write their parallel program on clusters as if they have shared memory access, specifying the layout of shared data at a higher level. However, it does not change the parallel programming model dramatically since programmers still need to write SPMD style parallel code and deal with the complexity of distributed arrays by identifying the specific data movement required for the parallel algorithm. The GA programming model forces the programmer to determine the needed locality for each phase of the computation. By tuning the algorithm such that the locality is maximized, portable high performance is easily obtained. Furthermore, since GA is a library-based approach, the programming model works with most popular language environments. Bindings are currently available for Fortran, C, C++ and python, and hence for those languages that are of interest when handling OpenMP.

GA programs distribute data in blocks to the specified number of processes. The current GA is not able to redistribute data. Before a region of code is executed, the required data must be gathered from the participating processes; results are scattered back to their physical locations upon completion. Since data distributions are simple, it is easy to compute the location of any data element. The implementation focuses on achieving very high efficiency in the data gathering and scattering phases. This approach is efficient if the regions of code are sufficiently large and the code is able to compute the gather and scatter sets only when really necessary. GA relies upon MPI to provide it with the execution context.

The most innovative idea of GA is that it provides an asynchronous one-sided, shared-memory programming environment for distributed memory systems. Both this shared memory abstraction, and the specific set of features GA offers, make it quite reasonable to translate OpenMP to GA. The traditional approach to implementing

OpenMP on distributed systems is based upon software DSM, which will transfer pages of data between memories when just a single element on that page has been modified, thereby potentially unnecessarily moving a considerable amount of data at synchronization points. GA provides a higher level of control, since the routines for gathering and scattering data can be used to specify precisely which data elements are to be transferred to which processor, and they also state when to do so. There are no "hidden" data transfers and there is no need to compare sets of changes to a page in memory. OpenMP maps computation to threads (just as GA maps computation to processes) and thereby indirectly specifies which data is needed by a given thread. This attribute makes it possible to translate OpenMP to GA. If the user has taken data locality into account when creating the OpenMP code, the benefits will be realized in the corresponding GA code.

3 The Translation Process

A careful study of OpenMP and GA routines showed that almost all of the OpenMP directives, library routines and environment variables can be translated into GA or MPI library calls at source level. Using GA and MPI together is not problematic since GA was designed to work in concert with the message passing environment. GA has the concept of shared data without explicit cooperative communication between processes. Coding for GA programs are similar to NUMA (non-uniform memory architecture) shared memory systems.

OpenMP parallel regions are transformed into GA program by invoking MPI_INIT and GA_INITIALIZE routines to initialize processes and the memory needed for storing distributed array data. Note too that the program only needs to call MPI_INIT and GA_INITIALIZE once in GA program for efficiency. Similarly, GA_TERMINATE and MPI_FINALIZE routines are called once to terminate the parallel regions.

The general approach to translating OpenMP into GA is to declare all shared variables in the OpenMP program to be global arrays in GA. Private variables can be declared as local variables that are naturally private to each process in a GA. If the parallel region contains shared variables, the translation will turn them into distributed global arrays in the GA program by inserting a call to the GA_CREATE routine. GA enables us to create regular and irregular distributed global arrays, and ghost cells (or halos) if needed. OpenMP FIRSTPRIVATE and COPYIN clauses are implemented by calling 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 computing processes respectively at run time. OpenMP provides routines to dynamically change the number of executing threads at runtime. We do not attempt to translate these currently since this would amount to performing data redistribution and GA is based upon the premise that this is not necessary.

In order to implement OpenMP parallel loops in GA, the generated GA program reduces the loop bounds according to specified schedule so as to assign work. Based on the calculated lower and upper bounds, and the array region accessed in the local code, each process in the GA program fetches a partial copy of global arrays via GA_GET, performs its work and puts back the modified local copy into global

locations by calling GA_PUT or GA_ACCUMULATE. The iteration set and therefore also the shared data must be computed dynamically For DYNAMIC and GUIDED loop schedules. We use GA locking routines to ensure that a process has exclusive access to code where it gets a piece of work and updates the lower bound of the remaining iteration set; the latter must be shared and visible for every process. However, due to the expense of data transfer in distributed memory systems, DYNAMIC and GUIDED schedules may not be as efficient as a static schedule, and it may not provide the intended benefits.

GA synchronization routines will replace OpenMP synchronizations. As OpenMP synchronization ensures that all computation in the parallel construct has completed, GA synchronization will do the same but will also guarantee that the requisite data movement has completed to properly update the GA data structures. GA locks and Mutex library calls are used to protect a critical section; we use them to translate the OpenMP CRITICAL and ATOMIC directives. The OpenMP FLUSH directive 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. GA provides the GA_SYNC library call for synchronization; it 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. Since GA works as a complement of MPI, and must be installed on a platform with GA, there is no problem invoking MPI routines in a GA program.

The translation of sequential program sections (serial regions outside parallel regions, OpenMP SINGLE, MASTER, and CRITICAL constructs) becomes non-trivial besides that of parallel regions. The program control flow must be maintained correctly in all processes so that some parts of the sequential section have to be executed redundantly by all processes. Subroutine/function calls in serial regions need to be executed redundantly if these subroutines/functions have parallel regions inside. We have identified three different strategies to implement the sequential parts: *master execution*, *replicated execution* and *distributed execution*.

In *master execution*, only the master process performs the computation, and gets/puts the global arrays before and after the computation. Exclusive master process execution of the sequential portion of programs invokes coherence issue of private data between master process and other processes; a broadcast operation is necessary after master process execution in order to achieve a consistent view of data.

In *replicated execution*, each process redundantly executes the same computation. At the end of computation, only one processor needs to update the global arrays using its own local copies, although all the processes need to fetch into local copies before the computation. The replicated execution approach has advantages of easy maintenance of the coherence of private data, and less data communication if a small number of shared variables are modified. But it has overhead of redundant computation and may cause more global synchronizations for shared memory updates and potentially too much data gathering. The approach could work when a sequential part computes mostly scalar data.

In *distributed execution*, the process that owns data performs the corresponding computation and keeps the computation in a certain order according to data dependency information. Each processor executes a portion of work of the sequential part according to constraints of sequential execution order. This may introduce

considerable synchronization. The distributed computation maximizes the data locality and minimizes the shared data communication, but may also require broadcasting of some data.

4 Related Work

OpenMP is not immediately implementable on distributed memory systems. Given its potential as a high level programming model for large applications, the need for a corresponding translation has been recognized. In our previous work, we have considered various strategies for helping the user improve the code prior to any strategy translating it for distributed execution, primarily by minimizing the amount of data that is shared [6].

A number of efforts have attempted to provide OpenMP on clusters by using it together with a software distributed shared memory (software DSM) environment [2, 3,8]. Although this is a promising approach, and work will continue to improve results, it does come with high overheads. In particular, such environments generally move data at the page level and may not be able to restrict data transfers to those objects that truly require it. There are many ways in which this might be improved, including prefetching and forwarding of data, general OpenMP optimizations such as eliminating barriers, and using techniques of automatic data distribution to help carefully place pages of data. The OMNI compiler has included additional data layout directives that help it decide where to place the pages of data in the various memories[8]. An additional approach is to perform an aggressive, possibly global, privatization of data. These issues are discussed in a number of papers, some of which explicitly consider software DSM needs [3, 4, 6, 9].

The approach that is closest to our own is an attempt to translate OpenMP directly to a combination of software DSM and MPI [5]. This work attempts to translate to MPI where this is straightforward, and to a software DSM API elsewhere. The purpose of this hybrid approach is that it tries to avoid the software DSM overheads as far as possible. While this has similar potential to our own work, GA is a simpler interface and enables a more convenient implementation strategy. Because it has a straightforward strategy for allocating data, it can also handle irregular array accesses, which is the main reason for retaining a software DSM in the above work. GA data has a global “home” but it is copied to and from it to perform the computation in regions of code; this is not unlike the OpenMP strategy of focusing on the allocation of work. For both models, this works best if the regions are suitably large. If the user is potentially exposed to the end result of the translation, we feel that they should be shielded as far as possible from the difficulties of distributed memory programming via MPI. GA is ideal in this respect as it retains the concept of shared data.

5 Conclusions and Future Work

This paper presents a basic compile-time strategy for translating OpenMP programs into GA programs. Our experiments have shown good scalability of the translated GA program in distributed memory systems, even with relatively slow interconnects. This

shared memory parallel programming approach introduces new overheads as it then must efficiently gather and scatter (potentially) large amounts of data before and after parallel loops. Our on-going work investigates the ability of the compiler to support the need for efficiency in these gather and scatter operations. We believe that recent advances in the MPI standard might enable GA to provide additional functionality that could increase the viability of this approach to parallel programming. We intend to explore this issue with our GA colleagues.

References

1. OpenMP Architecture Review Board, Fortran 2.0 and C/C++ 1.0 Specifications. At www.openmp.org.
2. C. Amza, A. Cox et al.: Treadmarks: Shared memory computing on networks of workstations. IEEE Computer, 29(2):18-28, 1996
3. A. Basumallik, S-J. Min and R. Eigenmann: Towards OpenMP execution on software distributed shared memory systems. Proc. WOMPEI'02, LNCS 2327, Springer Verlag, 2002
4. Chapman, B., Bregier, F., Patil, A. and Prabhakar, A.: Achieving High Performance under OpenMP on ccNUMA and Software Distributed Share Memory Systems. Currency and Computation Practice and Experience. Vol. 14, (2002) 1-17
5. R. Eigenmann, J. Hoeflinger, R.H. Kuhn, D. Padua et al.: Is OpenMP for grids? Proc. Workshop on Next-Generation Systems, IPDPS'02, 2002
6. Z. Liu, B. Chapman, Y. Wen, L. Huang and O. Hernandez: Analyses and Optimizations for the Translation of OpenMP Codes into SPMD Style. Proc. WOMPAT 03, LNCS 2716, 26-41, Springer Verlag, 2003
7. J. Nieplocha, RJ Harrison, and RJ Littlefield: Global Arrays: A non-uniform memory access programming model for high-performance computers. The Journal of Supercomputing, 10:197-220, 1996
8. M. Sato, H. Harada and Y. Ishikawa: OpenMP compiler for a software distributed shared memory system SCASH. Proc. WOMPAT 2000, San Diego, 2000
9. T.H. Weng and B. Chapman Asynchronous Execution of OpenMP Code. Proc. ICCS 03, LNCS 2660, 667-676, Springer Verlag, 2003