OLYMPIX - A PROGRAM PARALLELIZATION TOOL USING MPI ON COMPUTATIONAL GRIDS

K Vikram, Kumar Avijit and Sanjeev K Aggarwal
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur, 208016, India
email: ska@iitk.ac.in

ABSTRACT
Grid Computing aims at providing computing resources much in the same way as electricity is provided in a power grid. A grid is a large collection of computing resources that are managed by a resource manager to make it appear like a single powerful computing resource. The technology is still in its developing stages and has not yet matured to the point that a sequential task submitted to the grid can run on more than a single machine simultaneously. Automatic parallelization techniques have been proposed, that can parallelize sequential tasks into multiple threads that can be executed simultaneously with some communication between each other. Nevertheless, actual tools that apply these methods to computational grid, are still rare. In this paper, we discuss certain program parallelization techniques and propose a transformation to parallelize sequential programs for a grid. We discuss our tool, which is based on the SUIF. Our tool extends parallelization strategy of SUIF to convert sequential code into its semantic equivalent parallel code that can run on a grid.

Keywords: Grid Computing, Parallelizing Compilers, Dependence Analysis

1 Introduction

A grid comprises of heterogeneous resources over a wide area network. Grids aim at high performance computing and providing scope for innovative applications. Grid Computing is based on the *Grid Problem*[1] which is defined as the flexible, secure, co-ordinated resource sharing among dynamic collections of individuals, institutions and resources or *virtual organizations*. One of the main reasons for the popularity of grids is the rapid advancement in both speed and reliability of interconnect networks. Another reason for growth of grids is the realization that most workstations utilize a small fraction of their time in computations, leading to a lot of idle processor time that can be tapped.

Although the idea of using grids for high-performance computing is very attractive, there are a few issues. One of the important ones is that such usage of distributed resources would require breaking up of a program into smaller tasks that consume these distributed resources. Needless to say, this job of parallel programming is a difficult one that demands much more from the programmer. What would be most desirable is that the programmer would write code using a simple sequential model. The compiler that compiles this program would *parallelize* this code to generate binaries to be run on a grid. The large quantity of legacy codes cannot tap the power of a grid unless they are parallelized.

The importance and usefulness of tools that parallelize sequential programs thus cannot be neglected. There has been a lot of work related to this area and quite a few techniques for extracting the parallelism in programs have been proposed. These techniques though are not suitable for completely automating the parallelizing process. Semi-automatic parallelization is more common, where the programmer specifies certain parameters which help the compiler to generate parallel code. Parallelization for shared memory architectures are nevertheless more well studied and tools which achieve this also exist[2]. Parallelization for distributed memory architectures on the other hand are rarer. With the growing importance of large scale computational grids, this has to change. In this paper we look at some techniques that could be used to parallelize programs for running on a grid. We also discuss the details of our tool, OlyMPIx, built using these techniques. In what follows, in Section 2 we talk about the related work that has been going on in this area. Section 3 outlines in detail our approach towards building a usable tool for parallelization of sequential programs for running on a grid. In Section 4 we mention some important implementation details. Finally we end with some concluding remarks along with preliminary performance results in Section 5.

2 Related Work

In order to make use of as much computational resources as possible for executing a particular task, the task has to be intelligently divided into separate subtasks that can be executed on different processors. The most obvious way to go about this would be to let the programmer explicitly define the different subtasks in terms of what actions they perform. Quite a few standard libraries like PVM[3], MPI[4] and OpenMP[5] exist to allow the programmer specify communication and synchronization between subtasks. Although such an approach to writing parallel programs would be a powerful one, the programmer is burdened with all the work and has to figure out all the details.

One of the more popular methods has been the use of *data parallel languages*, *e.g.* HPF [6] that provide programming constructs such as parallel versions of sequential loops for writing parallel programs. The language also allows the user to specify the *data distribution* pattern, which indicates what part of the data would be present on which processor. The programmer in fact specifies a distribution pattern for a generic Cartesian grid of processors called a *template* that is mapped onto the actual processor ensemble by the compiler. In addition, [7] presents a survey of the various programming languages and systems that have been used for prototyping concurrent applications in general. Also, [8] discusses various parallel programming models and languages and the various criteria that should be met by these for them to be useful. [9] discusses certain software engineering issues such as code reuse, and outlines how Object Oriented technology can be used for this purpose.

A third, more interesting approach that we will focus on, is to let the programmer write normal sequential code and automatically convert it to a parallel version. There has been a lot of research related to this idea of automatic program parallelization and there are reasons to believe that it is promising [10]. [11] compares four different algorithms for parallelizing and argues that a semi-automatic approach where there is constant feedback from the programmer helps in generating better quality parallel code. Techniques for automatic parallelization for SMPs exist [12] and there are also usable prototype tools that apply these techniques [2]. On particular languages like Java, by exploiting its clean semantics, numerical code has been automatically parallelized to significantly enhance its performance [13].

Techniques for distributed memory systems concentrate mostly on data layout strategies applied after performing a *data dependence analysis*[14] of the program. [15] discusses a tool which automatically generates the data distribution directives in HPF from the parallel program. [16] also uses a similar idea, whereas [17] uses certain heuristics to compute an optimal data layout. [18] describes another methodology to automatically distribute data (array alignment).

However, for a grid like environment precise tools and techniques do not exist. Although grids fall under the category of generic distributed memory systems, some issues have to be dealt with differently. Communication overheads need to be modeled differently, which leads to a reanalysis of the problem of data distribution. Also, the cost of referencing shared variables would be quite different on a grid. The most important difference, however, is the highly dynamic nature of the grid due to a large variation in availability of resources during execution of a job on it. Also scaling issues need to be looked at carefully.

3 Our Approach

The major difference between parallelization for a shared memory model and for a distributed memory architecture is that the division of task among parallel threads may entail movement of data between distributed processors, which is not an issue in shared memory architectures. In short, parallelization for distributed memory architectures requires additionally, the distribution of data among threads.

The order in which task and data distribution is done could vary, and it could be even done simultaneously[19]. We have chosen to first distribute computation tasks, since we feel that an intelligent data distribution can be done better when the parallel control structure is known.

3.1 Task Distribution

The task distribution is based on extensive dependence analysis on the loops of the program. Since most of the running time of the program is centered around loops, most of the research has been towards parallelizing the loops of a program and we adopt the same strategy. We use the unimodular transformation theory[20] for the purposes of exposing coarse grain parallelism. Since we are targeting this for a grid like environment, we focus more on the coarse grain parallelism present in the program. The model of parallelism is *fork-join* and this is again quite suitable for a grid. In a grid the number of different processors could be many and the exact number might not be known a priori. Using such a model, we can divide the tasks into relatively many processors and the number of parallel threads associated with each loop would vary, keeping in mind the dynamic nature of a grid.

3.2 Data Distribution

We first discuss the basic approach adopted to distribute data from the master thread to the slave threads. We demonstrate how our data distribution minimizes the cost of communication between the simultaneous threads. We then go on to discuss how to further optimize on sending the data.

So far, the tool has parallelized the program for a shared memory architecture in which all the data needed by the slave nodes is automatically available with them. An additional issue of concern in parallelization for distributed memory architecture is that all the data needed by a slave node may not be present with it and may have to be transported from another slave node that computed its value.

olyMPIx resolves this issue by distributing all the data that is required by the slaves, and redistributing it at each fork point. In this approach, the slaves perform computations and report back the updated variables to the master. The master is solely responsible for distributing and redistributing the data required by the slaves. The data to be supplied to each slave is decided by the computation requirements of that slave. The data distribution is implemented by a simple template matching algorithm that we describe next.

The basic approach is to supply to a slave, only the data that it requires at its current fork instance. This is done by first finding out all the array references and global variables in the slave program. Let us assume for simplicity that the only statements in the slave code that may change the value of a variable are the assignment statements. The master distributes the right-hand side variables appearing in all the assignment statements in the slave's code. The slaves receive their data in precisely the same template as is used by the master to distribute the data. This template is actually the loop structure in which the slaves compute. Having finished the tasks, the slaves transfer all the variables appearing on the left hand side of the assignment statements to the master. This finishes a fork instance after which the master executes sequential code till it encounters the next fork instance. We demonstrate the data distribution with the help of the code below. The piece of code in Figure 1 shows the actual computation done by the slaves. `_my_id` is the identification number of the slave and `_my_nprocs` is the total number of slave procedures. The

```
for (priv_i = max(10*_my_id/_my_nprocs,0);
    priv_i <min(10*(_my_id+1)/_my_nprocs, 10);
                priv_i++)
    for (j=0;j<=9;j++)
           A[priv_i][j] = B[priv_i][j]*2;
```

Figure 1. Computation done by the slaves

code snippet in Figure 2 shows how a master thread distributes the data to the slaves that need it. `_my_id` is used to loop over all the slaves. The master sends all and only those variables that are eventually used during the computation at the slave threads.

```
for(_my_id = 0;_my_id < _my_nprocs -1; _my_id++)
 for (priv_i = max(10 * _my_id / _my_nprocs, 0);
       priv_i < min(10 *(_my_id + 1) /
       _my_nprocs, 10); priv_i++)
   for (j = 0; j <= 9; j++)
      MPI_Send(&B[priv_i][j], 1, MPI_Int, _my_id,
               0, MPI_COMM_WORLD);

for(_my_id = 0;_my_id < _my_nprocs -1;_my_id++)
 for (priv_i = max(10 * _my_id / _my_nprocs, 0);
       priv_i < min(10 * (_my_id + 1) /
       _my_nprocs, 10); priv_i++)
   for (j = 0; j <= 9; j++)
      MPI_Recv(&A[priv_i][j], 1, MPI_Int,
               _my_id, 0, MPI_COMM_WORLD);
```

Figure 2. The distribution pattern used by the master

3.2.1 Handling scalars inside a parallelized loop

So far, we have only looked at handling array definitions and uses inside the parallel loops. However, there may be assignments to scalar variables or uses of such variables. Such scalars need to be distributed to the slaves by the master. Finally. the index variable of the loop in the master thread needs to be updated after completion of the parallel function. We outline the following approach for this purpose:

- Identify all induction variables inside a parallelized loop.
- Slave executing last iteration sends the values of all induction variables.
- Master receives all induction variables at the end.

To elucidate the approach, let us consider the SUIF generated code shown in Figure 3 that handles updates to index variables. The slave executing the last iteration (indicated

```
static void main_0_func(my_id){
       int *i;
       int _priv_i;
       /* Loop using _priv_i */
       . . . . .
       if ( _my_id == my_nprocs )
                *i = _priv_i;
}
```

Figure 3. SUIF generated code for handling scalars

by `_my_id = my_nprocs`) updates the value of the variable `i`. This approach works well for shared memory architecture. Our tool transforms the code at the slave as given in Figure 4: The last slave sends the value of *_priv_i* to

```
static void main_0_func_put(my_id){
   int *i;
   int _priv_i;
   /*Put all other variables using _priv_i*/
   . . . . .
   if ( _my_id == my_nprocs -1 )
     *MPI_Send(&_priv_i, 1, MPI_Int,
       0, /* Send to master */
       0, MPI_COMM_WORLD);
}
```

Figure 4. Generated code for handling scalars

the master. Likewise, the master receives the variable after receiving all other results from the slaves. Other scalar variables inside the loop can also be handled in a similar way as outlined above for the index variables.

One drawback with the data distribution design proposed so far is that it does not handle multiple instances of sending/receiving the same data to a single processor.

```
for( int slave = 1; slave <= nslaves; slave++)
      for(int i=0; i<=I; i++)
           for(int j=0; j<=J; j++)
                  SEND(A[f(i,j)],slave);
```

Figure 5. Master thread distributing values of an array to slaves

```
for(slave=1; slave <= nslaves; slaves++)
    for(i=0; i<=10; i++)
        for(j=0; j<=10; j++)
            for(k=0; k<=10; k++)
               SEND(A[i+j+k][i+2j+k],slave);
```

Figure 6. The initial code fragment

For instance, consider the code (in Figure 5) in the master thread distributing values of an array to slaves:

The outer loop simply iterates over all the slave ranks while the inner two loops have been normalized for simplicity. The upper bounds `I` and `J` may depend on the value of slave. `f(i,j)` represents an affine function of `i` and `j`. A naive implementation of distributing data such as shown above may lead to sending a particular index of an array to a thread multiple number of times depending upon the existence of a solution to the equation: f(i_1,j_1) = f(i_2,j_2) subject to the constraints:

$$0 \leq i_1, i_2 \leq I, 0 \leq j_1, j_2 \leq J \tag{1}$$

The basic idea is to pick up points (i, j) from the iteration space such that no two points yield the same value of $f(i, j)$. This can be done by solving (1) to partition the iteration space into equivalence classes. In the case when no two tuples of the iteration space yield the same value of $f(i, j)$, all the equivalence classes will comprise of single points. The next step is to iterate over all the equivalence classes choosing any one point belonging to the class, and send/receive the data corresponding to that tuple. We illustrate the point with the aid of an example. The initial code fragment is as in Figure 6.

Dependence analysis done considering only a single dimension of A at a time yields the equivalence classes for that dimension as planes in the (i, j, k) space. The final equivalence classes are then found out by taking the intersection of the following two sets of planes:

$$i + j + k = P_1, i + 2j + k = P_2; P_1, P_2 \in [0, 30] \tag{2}$$

In the above case, the equation $j = C, i+k = P_1 - C$ gives the equivalence classes for various values of C and P_1. The number of loop iterations finally required are, therefore the number of valid values that C can take, plus any points on the boundary that might have been left out. The code in Figure 7 shows the optimized version for the program in Figure 6. The reader can convince himself that the above loop will actually cover all the values of A without any repetition.

```
for(slave=1; slave<=nslaves; slave++)
{
     for(k=0; k<=10; k++)
          for(j=0; j<=10; j++)
               SEND(A[0+j+k][0+2j+k],slave);
     for(k=0; k<=10; k++)
          for(i=0; i<=10; i++)
               SEND(A[i+10+k][i+20+k],slave);
}
```

Figure 7. The optimized code

The above approach may further be optimized for performance. For instance, instead of sending individual data elements while data distribution, an aggregate of these values can be sent to the slaves. This reduces the performance overhead of making an MPI call to send data.

4 Implementation Details

The transformations that have been discussed in Section 3 have been implemented as SUIF [21] passes, details of which are discussed in Section 4.1. SUIF is a compiler infrastructure that provides an intermediate representation for representing programs at an appropriate level of detail. It enables writing new passes that work on the intermediate representation of the program which can then be translated back to the source by SUIF back-end.

4.1 The data distribution pass

Like all passes in SUIF, the pass that we wrote takes its input as a "SUIF" file and writes its output to another "SUIF" file. The input is the parallelized version that is obtained by running appropriate unimodular transformations[20] on the input code. Since the SUIF parallelizer already has a very similar transformation written as a series of passes, we make use of the data dependence analysis and control structure analysis done by SUIF for our purpose. Assuming that the task distribution has already been done and calls to the SUIF runtime library have been added, we now modify this program to generate our parallelized code for distributed memory architectures. This entails two different tasks. One is to add in code for the actual data distribution. The other is to replace SUIF runtime library calls with our library calls. The data distribution code is generated by using the *slave* code fragment as a template. Detailed analysis is done on this piece of code to figure out the exact data requirement of the slave. It is important to note that although this information could have been found out by earlier intermediate structures, we keep the design of our system general in the sense that our transformation would work on any program that has been parallelized for shared memory architectures. We therefore do not assume that the program necessarily has affine array indices and has been parallelized by using unimodular transforms.

We run through the code fragment of the template to isolate the array variables that are being used in the computation. Our assumption that arrays are the main storehouses of input data for a compute intensive program is a fairly realistic one. We first categorize each array reference as either an *lvalue*, or a *rvalue*. For such purposes SUIF provides procedures to handle the intermediate format and read from an instance of it. Subsequently detailed dependence analysis is done on these array references, depending on which loop they are present in. The built in dependence library of SUIF is used for performing this analysis. The library functions provide us with the dependence vectors which are used to construct a virtual dependence graph. As explained in Section 3.2 this dependence graph is broken into disjoint subgraphs which help us in deriving the exact array indices for which the array data should be sent.

With the help of the `builder` library of SUIF we then form the appropriate loops which iterate over precisely these indices. Using these techniques, the template code is used to generate four different procedures. One in the master to distribute data to all the slaves and a corresponding one in the slave to receive this data. The other two send and receive data respectively after the computation in the slave is over. Again the `builder` library is used for this purpose with occasional tweaking of the SUIF structures directly where necessary.

The MPI[4] library is used for the `send` and `receive` calls for sending and receiving data. MPI was specifically chosen for its standardized interface and easy portability. This would also help us to extend our tool to support MPICH-G[22] which is a message passing interface more suitable for grid like environments.

5 Conclusion and Future Work

In this paper we have discussed the importance of automatic program parallelization, especially for grid like environments. Our initial performance results are promising and even better results could be achieved by further optimizations. We show below the performance results for a compute-intensive parallel program. The sequential program consisted of three 100x100 arrays containing `float` values. The main loop consisted of compute-intensive operations on the arrays in a manner similar to that of matrix multiplication. The sequential program was first parallelized using olyMPIx. The parallel program was run on a grid and the number of available processors was varied from 2 to 6. Figure 8 shows the time taken by the parallel program to run on the grid. The first case with number of threads equal to 1 is the time taken by the sequential program. With $n \geq 2$ number of threads, the number of threads doing the actual computation is actually $n - 1$. This is because, the master thread is responsible only for dispatching data between slaves and does not share parallel iterations with the slaves. The speedup obtained from parallelization in this case, is quite close to the theoretical ideal value. The time taken by the parallel program with $n = 2$ is slightly more than that required by the sequential version. This is because the program is actually not parallelized in this case as the computation is still done by a single slave. In addition, in this case, there is an extra overhead of moving data between master and the slave which makes it costlier than the sequential version.

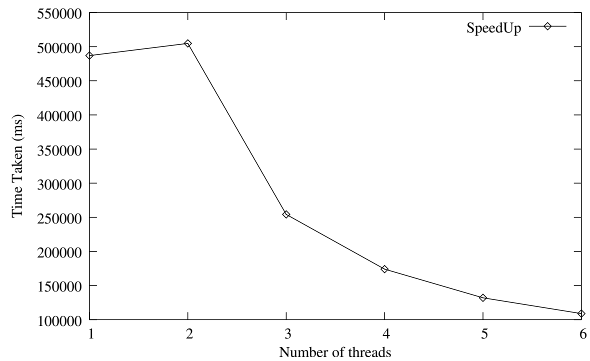

Figure 8. Speedup obtained by parallelization for grid

Improving the performance could be done in quite a few ways. For the same messages that are being sent to different processors, a broadcast could be used instead of multiple unicasts. This would definitely reduce some message passing overhead. Also, with the mechanism of MPI being a *push* mechanism certain things such as initiation of data transfer from the slaves would not be possible. With a *pull* mechanism of message passing this could be done to reduce some of the static analysis that the transformation pass is supposed to do. Also, with our implementation following an SPMD model, the same program runs on all nodes. Thus the slaves in the generated parallel program use more memory than necessary. This could be changed by dynamically allocating arrays rather than doing so statically. Another feature that could be added to the system is that of fault-tolerance. If one of the slaves goes down and does not return the results of the computation it was assigned, the master should assign this subtask to another available slave. More importantly, if the master itself dies, then a second *shadow* master should take over and continue the task. We are working on incorporating these enhancements into our tools.

References

[1] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, *2150*:1–71, 2001.

[2] The SUIF 1.x compiler - software distribution. http://suif.stanford.edu/suif/suif1/index.html.

[3] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, *2*(4):315–340, 1990.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[5] OpenMP: A proposed industry standard API for shared memory programming. OpenMP white paper, October 1997.

[6] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: design, implementation, and performance results. In *Proc. 1993 ACM/IEEE Conference on Supercomputing*, pages 351–360. ACM Press, 1993.

[7] Wilhelm Hasselbring. Programming languages and systems for prototyping concurrent applications. *ACM Comput. Surv.*, *32*(1):43–79, 2000.

[8] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, *30*(2):123–169, 1998.

[9] J.-M. Jezequel. An object-oriented framework for data parallelism. *ACM Comput. Surv.*, *32*(1):31, 2000.

[10] Michael Frumkin, Michelle Hribar, Haoqiang Jin, Abdul Waheed, and Jerry Yan. A comparison of automatic parallelization tools/compilers on the SGI origin 2000. In *Proc. IEEE/ACM SC98 Conference*, page 39. IEEE Computer Society, 1998.

[11] N. Mukherjee and J. R. Gurd. A comparative analysis of four parallelisation schemes. In *Proc. 13th International Conference on Supercomputing*, pages 278–285. ACM Press, 1999.

[12] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar. Automatic parallelization for symmetric shared-memory multiprocessors. In *Proc. 1996 Conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1996.

[13] Pedro V. Artigas, Manish Gupta, Samuel P. Midkiff, and Jose E. Moreira. Automatic loop transformations and parallelization for java. In *Proc. 14th International Conference on Supercomputing*, pages 1–10. ACM Press, 2000.

[14] Kleanthis Psarris and Konstantinos Kyriakopoulos. The impact of data dependence analysis on compilation and program parallelization. In *Proc. 17th Annual International Conference on Supercomputing*, pages 205–214. ACM Press, 2003.

[15] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, *20*(4):869–916, 1998.

[16] Jordi Garcia, Eduard Ayguade, and Jesus Lebarta. A novel approach towards automatic data distribution. In *Proc. 1995 ACM/IEEE Conference on Supercomputing (CDROM)*, page 78. ACM Press, 1995.

[17] Paul Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In *Proc. 7th International Conference on Supercomputing*, pages 175–184. ACM Press, 1993.

[18] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–28. ACM Press, 1993.

[19] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proc. 1989 ACM/IEEE Conference on Supercomputing*, pages 637–646. ACM Press, 1989.

[20] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, *2*(4):452–471, 1991.

[21] S. Amarasinghe, J. Anderson, M. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, 1995.

[22] Ian Foster and Nicholas T. Karonis. A grid-enabled MPI: message passing in heterogeneous distributed computing systems. In *Proc. 1998 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–11. IEEE Computer Society, 1998.