Improving Separation of Concerns in the Development of Scientific Applications

S. M. Sadjadi*, J. Martinez, T. Soldo, L. Atencio
School of Computing and Information Sciences
Florida International University, Miami, FL, U.S.A
{sadjadi,ftrig001}@cs.fiu.edu

R. M. Badia, J. Ejarque
Barcelona Supercomputing Center
Barcelona, Spain
{rosa.m.badia, jorge.ejarque}@bsc.es

Abstract

High performance computing (HPC) is gaining popularity in solving scientific applications. Using the current programming standards, however, it takes an HPC expert to efficiently take advantage of HPC facilities; a skill that a scientist does not necessarily have. This lack of separation of concerns has resulted in scientific applications with rigid code, which entangles non-functional concerns (i.e., the parallel code) into functional concerns (i.e., the core business logic). Effectively, this tangled code hinders the maintenance and evolution of these applications. In this paper, we introduce Transparent Grid Enabler (TGE) that separates the task of developing the business logic of a scientific application from the task of improving its performance. TGE achieves this goal by integrating two existing software tools, namely, TRAP/J and GRID superscalar. A simple matrix multiplication program is used as a case study to demonstrate the current use and capabilities of TGE.

Keywords

Grid Enablement, Transparent Shaping, GRID superscalar

1. Introduction

The advent of cluster and grid computing has created a remarkable interest in high performance computing (HPC) both in academia and industry, especially as a solution to complex scientific problems (e.g., hurricane path prediction). To efficiently utilize the underlying HPC facilities using the current programming models and tools, however, a scientist is expected to develop complex parallel programs; a skill that she might not necessarily have and is better done by an HPC expert.

Current standards for cluster and grid programming such as MPI [8], OGSA [9], and WSRF [10] (and their implementations such as MPICH2 [11], Globus Toolkit [12], Unicore [13], and Condor [14]; to name just a few) have provided scientists with higher levels of abstraction. Noteworthy, these approaches have been successful in hiding the heterogeneity of the underlying hardware devices, networking protocols, and middleware layers from the scientist developers. However, the scientists are still expected to develop complex parallel algorithms and programs. Moreover, as the code for parallel algorithms would typically crosscut the code for business logic of the application, the resulting code will be an entangled code that is difficult to maintain and evolve.

In this paper, we introduce Transparent Grid Enabler (TGE) that addresses these problems by enabling a *separation of concerns* in the development and maintenance of the non-functional concerns (i.e., the parallel code) and the functional concerns (i.e., the business logic) of scientific applications. TGE achieves this goal by integrating two existing programming tools, namely, a Grid framework, called *GRID superscalar* [2], and an adaptation-enabling tool, called *TRAP/J* [6].On one hand, GRID superscalar enables the development of applications for a computational Grid by hiding details of job deployment, scheduling, and dependencies and enables the exploitation of the concurrency of these applications at runtime. On the other hand, TRAP/J supports automatic weaving of alternative parallel code (including the corresponding calls to GRID superscalar runtime) into the sequential code developed by the scientist.

TGE increases the level of modularity of code by separating crosscutting grid related code from the business logic of the application. This allows scientists to continue focusing only on the core logic of the scientific applications, leaving the parallel code and its complexity to the HPC experts. In TGE, the grid enablement or weaving of parallel code into the original application is called to be transparent, because all the grid enablement process occurs automatically with no manual modifications to the business logic of the application and hence "transparent" to the scientist and her sequential code. This way, TGE supports transparent grid enablement of existing scientific applications also.

The rest of this paper is organized as follows. In Section 2, we provide a short background on GRID superscalar and TRAP/J. In Section 3, we introduce a simple case study, called "Matmul", which is a matrix multiplication program. In Section 4, we show how TGE works by adapting Matmul to run on a computational grid. In Section 5, we provide some experimental results and demonstrate the speedup gained because of grid enablement. In Section 6, we discuss some related works and in Section 7, we provide some future research directions. Currently, TGE enables only static grid enablement of Java programs by means of configuration files at startup time. We are planning to

provide dynamic grid enablement as well as self-management behavior to scientific applications at run time. Finally, we finish the paper in Section 7 after providing some concluding remarks.

2. Background

Transparent grid enablement is achieved by the combination of TRAP/J and GRID superscalar. Therefore, as a first step, we present a brief background information about both technologies. For more detail, please refer to the references.

2.1 GRID superscalar

Inspired by the *superscalar* processors, GRID superscalar provides an easy programming paradigm for developing parallel programs [2]. Similar to superscalar processors that provide out-of-order and parallel execution of machine instructions by bookkeeping their dependencies, GRID superscalar provides parallelism to the functions of a program written in a high-level programming language such as Java. Using GRID superscalar, a sequential scientific application developed by a scientist is dynamically parallelized in a computational Grid. GRID superscalar hides the details such as resource mapping, staging input data files, cleaning temporary data files, task deployment, task scheduling, exploiting instruction-level parallelism, and exploiting data locality. We note that for many of its responsibilities, GRID superscalar depends on other grid computing toolkits such as GT4 [12], Condor [14], and others.

2.2 TRAP/J

TRAP/J is a tool that enables static and dynamic adaptation in Java programs at startup and runtime, respectively [6]. It consists of two GUI-based interactive tools as follows: (1) *Generator*, which generates an adapt-ready version of an existing application by inserting generic hooks into a previously selected subset of classes in the application; and (2) *Composer,* which allows insertion of new code at the generic hooks both at startup or runtime. We note that only the pre-selected classes are capable of being adapted and they are called *adaptable* classes. Adaptable behavior is provided through alternative implementation of adaptable classes, which are called *delegate* classes. To replace alternative parallel algorithms developed using the GRID superscalar codes, we use the Generator to make the classes with sequential code adaptable, and then we use the Composer to weave in the parallel code.

3. Case Study: Matmul

Our case study is a simple application, called Matmul, which is a matrix multiplication program written in Java. It uses a sequential matrix multiplication algorithm, which computes C = A.B, where A, B, and C are matrices of size NxN. It uses the classic algorithm of "rows by columns" multiplication. This algorithm involves $O(N^3)$ operations.

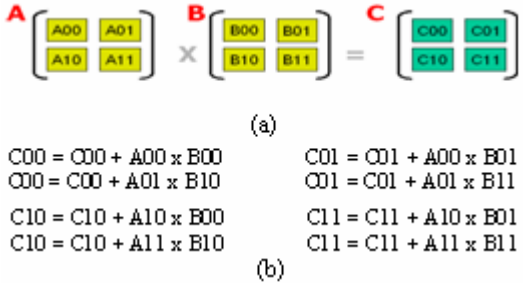

Figure 1: Hyper-Matrix Multiplication.

We will use TGE to make this application grid enabled. First, we use GRID superscalar to develop alternative hyper-matrix multiplication algorithms by splitting the original matrices into a number of sub-matrices or blocks as shown in Figure 1 (a). Then we multiply these sub-matrices accumulatively as shown in Figure 1 (b). GRID superscalar will exploit the task-level parallelism by resolving the dependencies of the tasks as shown in Figure 1 (b). Therefore, instead of just one task as in the original approach, using hyper-matrix multiplication and GRID superscalar, up to 4 tasks can be active at the same time. Similarly, if we split the matrix into 9 blocks, then up to 9 tasks can be executed at the same time and so on and so forth.

4. Grid Enablement of Matmul

We begin with the snippet code of the simple sequential matrix multiplication program that is shown in Figure 2. The bold method method in Figure 2 performs a conventional row by column matrix multiplication. The statement underlined saves the result of the multiplication in the specified file.

```
public static void main(String[] args)
{       . . .
Multiply_Matrices(size, args[1], args[2],
args[3]);
//args[] contains the names of the files
//containing the input matrices
}
public  static  void  Multiply_Matrices(int
size, fileC, fileA, fileB)
{ Block A = new Block(fileA, size);
  Block B = new Block(fileB, size);
  Block C = new Block(size);
  C.Multiply(A,B);
  C.blockToDisk(fileC);
}
```

Figure 2: Original Matrix Multiplication Code

In order to run this application on the grid we will use TRAP/J to weave in the parallel code developed at startup time into this application and use GRID superscalar to run the grid enabled adapted program. We select the Multiply_Matrices method to become adaptable since this method has been identified as the computationally intensive part of the original application. Therefore, a delegate class is developed that re-implements this method using the hyper matrix multiplication algorithm and GRID superscalar.

As shown in Figure 1 (b), since the calculation of each block in the resulting matrix (*C*) is independent of the other ones, they can be executed in parallel, potentially on different processors of a grid computing environment. Figure 3 displays the code for a delegate class, Matmul_Del.java, that was implemented for this case study and includes the Multiply_Matrices method. The beginning of the method (not shown here for simplicity) takes care of splitting the original matrix operands *A* and *B* into blocks, creating files where each block is saved, and creating the files that will store the result of matrix multiplication for each block.

```
public class Matmul_Del implements Delegate
    Interface
{  public static void Multiply_Matrices(int
    size, fileC, fileA, fileB)
  { . . .
   GSMaster.On();
   for(int i=0;i<num_of_pieces;i++)
   {  //Split in 4 pieces-
     for(int j=0; j<num_of_pieces;j++)
     {
      for(int k=0; k<num_of_pieces;k++)
      {  //Sending to Grid
         Matmul.multiply_acc(C[i][j],
               A[i][k],B[k][j],size/
               num_of_pieces);
      }
    GSMaster.Off();
     . . .
    MergeFiles();
}
```

Figure 3: Delegate Class for Multiply_Matrices method

The underlined section of the code calls the matrix multiplication method, Matmul.multiply _acc() for each pair of corresponding blocks. This is the method that allows for parallelism by being executed in separate tasks (possibly running on different nodes). In order for GRID superscalar to know that Matmul.multiply_acc() is the method to be deployed on worker nodes, several steps must be taken. First, an IDL file must be created as shown in Figure 4 to specify the signature of this method.

```
interface MATMUL
{    void multiply_acc(inout File f3,
in File f1, in File f2, in int size);
};
```

Figure 4: Matmul IDL file

This part is much like a CORBA IDL file that is used to generate stubs and skeletons to be used for a remote procedure call. The IDL uses the special keywords “in”, “out”, and “inout” to specify the type of the parameters to be read, written or both, respectively. Using this IDL as input to GRID superscalar, we generate the “worker” versions of Matmul; and using the selected adaptable method as input to TRAP/J, we generate the “master” version of Matmul.

When we execute the master program, it calls the Multiply_ Matrices method, which will be intercepted by the TRAP/J runtime and will forward the control to the code in Figure 3; effectively the parallel code will be executed instead of the original sequential code. As a first step, GRID superscalar will be started with a call to GSMaster.On(), basically to initialize resources in the grid, like for example Globus services. Later, each multiplication of the sub-matrices will be sent to the nodes in the grid using the Matmul_multiply_acc(…). After all the calls to this method for all the multiplications are done, GRID superscalar is disabled by the call to GSMaster.Off(). Finally GRID superscalar runtime will collect the results using the mergeFiles() method, which is in charge of merging the individual output files obtained from the different block multiplications into one matrix file representing the result of the matrix multiplication.

The class Matmul on which the static method Multiply_Accumulative() is invoked is actually provided by GRID superscalar when deploying the application. Basically what it does is to call the GRID superscalar runtime in order to execute the method, Multiply_Accumulative() already defined in the IDL we created at startup. As mentioned before, all the issues related to file handling, concurrency problems, and interactions with the grid (middleware like Globus in this case) are handled by GRID superscalar.

Finally, the obtained grid-enabled application offers the choice for the user to choose among different alternative parallel-computing algorithms; for example, choosing between an algorithm which uses 4 blocks or 9 blocks. The decision of choosing one algorithm or another can be made based on the number of resources available and therefore, taking advantage of the grid infrastructure properly. Moreover, this new grid-enabled application is transparent to the user in the sense the way it was originally executed remains the same.

5. Experimental Results

The case study we discussed on the sections above left us with some interesting results that we present in this section.

First, we should point out that even though our approach takes advantage of parallel programming when using the computational grid, we also face the problem of delays caused by the network traffic, coordination of tasks, and the middleware software services used (Amdahl's law). Therefore, it is reasonable to predict that when the matrices to multiply present a relatively small dimension, the original sequential application will perform faster than the grid-enabled one. As the matrix size increases we will be able to see that this difference in time shortens progressively.

Matrix Size (*N*)	Sequential (ms)	Parallel with 4 blocks (ms)	Speedup (S/P)
144	674	61512	0.010957212
288	2031	66096	0.030728032
576	9527	69365	0.137345924
1152	62269	172787	0.360380121

Table 1: Initial time results

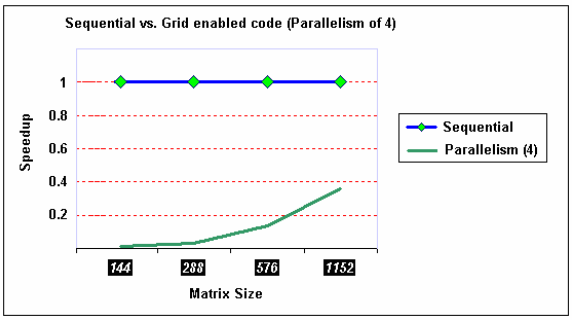

Figure 5: Speedups of the experiments

Table 1 shows the initial experiments we ran, and there you can compare the results obtained by both the original sequential program versus our approach using four nodes of the grid and therefore having a level of parallelism of 4. In this first set of experiments, we noticed that the sequential code always ran faster than the grid-enabled one; however, as the matrix size increased, we could notice that the difference in time between the two approaches became smaller and smaller.

One of the main problems we had by then with the performance was due to the way GRID superscalar works. The method GS_Off(), mentioned in section 4 and in figure 3, is in charge of freeing resources and deleting temporary files after finishing the calls to the grid method and since all the data is distributed along the nodes, then a cleanup was needed everywhere causing our application to take extra time.

Since we wanted to get a more optimized grid-enabled application, we took the GRID superscalar source code and optimized it, removing the cleanup but keeping the main functionality so that we can still get consistent results. Applications that benefit from HPC are usually scientific applications like, for example, those related to hurricane prediction and monitoring. In such cases, temporary left over data is irrelevant if the application provides us with the correct results quickly. Therefore, the approach we took at this stage seems proper.

With this optimization in hand, we also decided to implement a new algorithm in which we handled a parallelism of 9. Due to infrastructure reasons at the time, the number of nodes available for this experiment was at most 6. The results in terms of times consumed for this set of new experiments are shown in Figure 6 and 7.

Matrix Size (*N*)	Seq. (ms)	Par. w/ 4 blocks and 2 workers (ms)	Par. w/ 4 blocks and 4 workers (ms)	Par. w/ 9 blocks and 6 workers (ms)
144	5576	79221	57656	145331
288	14934	86259	62013	146744
576	44755	108107	78096	148240
1152	19318	176464	133058	176464
2304	79837	643925	441891	474215

Table 2: Final time results

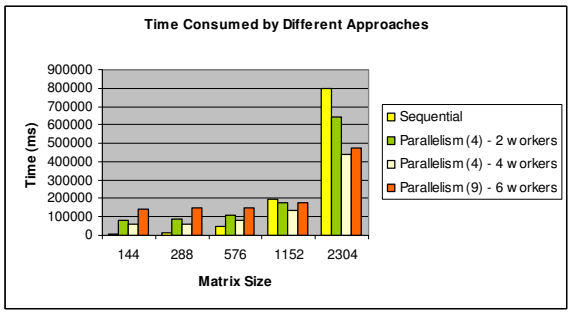

Figure 6: Chart of the times for each approach

With the results (time in ms) obtained in Table 2 and Figure 6 we were able to build a table and a graph showing the Speedups of each algorithm as shown in table 3 and figure 7.

In Figure 7 we see that as the matrix size increases, the speedup improves and finally when the size of the matrix is 1152 (number of rows = number of columns = 1152), all of the algorithms for the grid-enabled application perform better than the original sequential application. Furthermore, when the size of the matrix is 2304, all of the algorithms perform even much better than the sequential one. As a result, we notice that the algorithm with best performance is the one that uses parallelism of 4 and 4 nodes which for a matrix of size 2304 performs almost twice faster than the

sequential one. This is because we have more CPU power and we are using all of it because of the parallelism of 4. Besides that, having only 4 nodes, reduces the number of file transfers among the nodes, which in turn reduces time of execution.

Matrix Size	Seq	Parallelism (4) 2 workers	Parallelism (4) 4 workers	Parallelism (9) 6 workers
144	1	0.0703858	0.09671153	0.03836759
288	1	0.1731298	0.24082048	0.10176907
576	1	0.4139879	0.57307673	0.30190907
1152	1	1.0947502	1.45187813	1.09475020
2304	1	1.2398462	1.80670799	1.68355704

Table3: Speedups of each approach

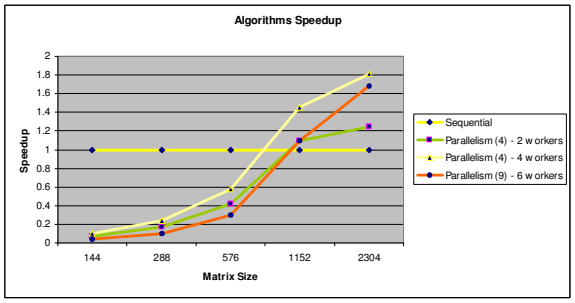

Figure 7: Speedups of each approach

We emphasize that the experiments are part of our ongoing research activities and by no means they are meant to be representative and conclusive with respect to providing a quantitative metric for speedup of sequential applications. The main purpose of these experiments is to show that we were able to use the current prototype of TGE to transparently adapt an application to run on a grid computing environment.

6. Related Work

Other approaches that enable programming parallel applications for computational Grids are Satin, HOCS, ProActive or ASSIST.

Satin [7] is a Java based programming model for the Grid which allows to explicitly expressing divide-and-conquer parallelism. Satin uses marker interfaces to indicate that certain invocation methods need to be considered for potentially parallel (spawned) execution. Moreover, synchronization is also explicitly marked whenever it is required to wait for the results of parallel method invocations.

HOCS [5] is a component oriented approach based on a master-worker schema. Higher-Order Components (HOCs) express recurring patterns of parallelism that are provided to the user as program building blocks, pre-packaged with distributed implementations.

ASSIST [1] is a programming environment aimed at providing parallel programmers with user-friendly, efficient, portable, fast ways of implementing parallel applications. It includes a skeleton based parallel programming language (ASSISTcl, cl stands for coordination language) and a set of compiling tools and run time libraries. The ensemble allows parallel programs written using ASSISTcl to be seamlessly run on top of workstation networks supporting POSIX and ACE (the Adaptive Communication Environment, which is an extern, open source library used within the ASSISTcl run time support).

ProActive [3] is a Java GRID middleware library for parallel, distributed and multi-threaded computing. With a reduced set of simple primitives, ProActive provides a comprehensive API to simplify the programming of Grid Computing applications: distributed on Local Area Network (LAN), on clusters of workstations, or on Internet GRIDs. ProActive is only made of standard Java classes, and requires no changes to the Java Virtual Machine, no preprocessing or compiler modification, leaving programmers to write standard Java code. Architected with interception and reflection, the library is itself extensible, making the system open for adaptations and optimizations. Current implementation is focusing of the CoreGRID NoE specification of the Grid Component Model (GCM) [4].

None of the above mentioned approaches provide an explicit separation of concerns identifying separate tasks for scientist developers and HPC expert developers. TGE can be extended to use these works instead or in complement to GRID superscalar and can be used as an enabler for supporting interoperation among the above mentioned approaches.

7. Future Work

As we mentioned before, we have been able to achieve *static* adaptation. Our next task will be to extend TGE in support of more autonomic behavior and include adaptation at runtime (dynamic) in response to high level system policies such as the addition of more nodes to the grid, process scheduling, etc, or application level policies such as different blocking algorithms, faster algorithms, etc.

At present, dynamic adaptation of Java programs with TRAP/J has been achieved and tested. However, running an application in a grid environment inherently introduces a lot more challenges than just running the program in one node or virtual machine. If we take Matmul as an example, we can clearly see that switching the blocking algorithm dynamically requires us to save the present state of

calculations, adapt it to the new algorithm, and continue executing.
Furthermore, moving towards building a more autonomic self adapting and self configuring system, we can take TGE to provide context-aware adaptation. In other words, by invoking the Globus Toolkit monitoring service we can keep track of the state of the runtime environment and retrieve information about resource allocation, scheduling, etc.

8. Conclusion

In this paper we have presented an innovative approach to transparent grid-enablement of scientific applications. We achieved this goal by combining TRAP/J and GRID superscalar. Each tool provided us with the necessary features for transparent software adaptation from a sequential code to a grid-enabled one as Figure 8 briefly sums up.

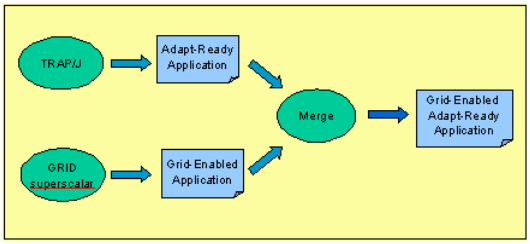

Figure 8: TGE Flow Diagram

The matrix multiplication shown as a case study in this paper is just a simple example to show how our approach works. In fact, this matrix multiplication, for example, could be just one part of a whole application and be the portion of the code that consumes most of the execution time and in that sense, applying our approach would considerably benefit the whole application's performance. In a similar fashion, we could take an existing application and just modify the part of the algorithm that is in charge of the most cpu utilization and just parallelize that logic without having to modify the total original code.

Another important issue to mention is that the optimization discussed in the paper is not targeted only to improve the performance of our case study, in fact, since this optimization has been done to the Grid Superscalar library, any application built from now on will benefit for these improvements. Moreover, even though we took Java as the programming language to describe our case study, other programming languages could be used following the same approach since for example C and Perl are also supported by Grid Superscalar.

Finally, we are aware that we cannot guarantee that in all applications we will be able to separate the parallelism of a portion of the algorithm from the business logic of it; however, there many existing applications that do offer this facility.

9. Acknowledgement

This work was supported in part by IBM (SUR and Student Support awards), the National Science Foundation (grants OCI-0636031, REU-0552555, and HRD-0317692), the Spanish CICYT (contract TIN2004-07739-CO2-01), and the BSC-IBM Master R&D Collaboration agreement. This work is part of the Latin American Grid (LA Grid) project.

References

[1] Marco Aldinucci, Massimo Coppola, Marco Danelutto, Marco Vanneschi, and Corrado Zoccolo. *Assist as a research framework for high-performance grid programming environments*. In Jose C. Cunha and Omer F. Rana, editors, Grid Computing: Software environments and Tools. Springer-Verlag, 2004.

[2] Rosa M. Badia, Raül Sirvent, Jesus Labarta, and Josep M. Perez. *Programming the GRID: An Imperative Language Based Approach*. book chapter in Engineering the Grid, Section 4, Chapter 12 , January 2006.

[3] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Programming, Composing, Deploying for the Grid* (the reference to be used to cite ProActive), in "GRID COMPUTING: Software Environments and Tools", Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.

[4] CoreGRID Deliverable D.PM.02, 2006, Proposal for a Grid Component Model.

[5] Sergei Gorlatch and Jan Dünnweber. *From Grid Middleware to Grid Applications: Bridging the Gap with HOCs*. In Future Generation Grids, Springer Verlag, 2005.

[6] S. Masoud Sadjadi, Philip K. McKinley, Betty H.C. Cheng, and R.E. Kurt Stirewalt. *TRAP/J: Transparent generation of adaptable Java programs.* In Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus, October 2004.

[7] Rob van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. *Satin: Simple and efficient Java-based grid programming*. Scalable Computing: Practice and Experience, 6(3):19-32, September 2005.

[8] http://www-unix.mcs.anl.gov/mpi/

[9] http://www.globus.org/ogsa/

[10] http://www.globus.org/wsrf/

[11] http://www-unix.mcs.anl.gov/mpi/mpich2/

[12] http://www.globus.org/toolkit/

[13] http://www.unicore.org/

[14] http://www.cs.wisc.edu/condor/