Journal of Applied Computing Research, 1(2):84-94
July-December 2011

Flexible Management on BSP Process Rescheduling: Offering Migration at Middleware and Application Levels

Lucas Graebin, Rodrigo da Rosa Righi
Universidade do Vale do Rio dos Sinos. Av. Unisinos, 950, 93.022-000, São Leopoldo, RS, Brasil.
lgraebin@acm.org, rrrighi@unisinos.br

Abstract. This article describes the rationales for developing jMigBSP - a Java programming library that offers object rescheduling. It was designed to work on grid computing environments and offers an interface that follows the BSP (Bulk Synchronous Parallel) style. jMigBSP's main contribution focuses on the rescheduling facility in two different ways: (*i*) by using migration directives on the application code directly and (*ii*) through automatic load balancing at middleware level. Especially, this second idea is feasible thanks to the Java's inheritance feature, in which transforms a simple jMigBSP application in a migratable one only by changing a single line of code. In addition, the presented library makes the object interaction easier by providing one-sided message passing directives and hides network latency through asynchronous communications. Finally, we developed three BSP applications: (*i*) Prefix Sum; (*ii*) Fractal Image Compression (FIC) and; (*iii*) Fast Fourier Transform (FFT).They show our library as viable solution to offer load balancing on BSP applications. Specially, the FIC results present gains up to 37% when applying migration directives inside the code. Finally, the FFT tests emphasize strength of jMigBSP. In this situation, it outperforms a native library denoted BSPlib when migration facilities take place.

Keywords: Bulk Synchronous Parallel, Rescheduling, Java, Adaptation, Object migration, Grid computing.

Introduction

Load balancing is a key issue for getting performance on heterogeneous and dynamic distributed environments (El Kabbany *et al.*, 2011). It can be enabled by rescheduling execution entities like processes, tasks or objects. This technique is useful to migrate entities for executing faster on lightly-loaded resources and/or approximating those ones that communicate frequently. Basically, migration facility can be offered at application or middleware levels (Pontelli *et al.*, 2010). The former idea can use explicit calls in the source code while the second represents an extension of the programming library for providing both transparent and effortless mechanism of migration at user's point of view. Considering the last statement, the programming library acts on middleware that control which objects will migrate, the moment of that as well as the selection of the destination nodes. Research on rescheduling topic includes the definition of unified metrics for acting in response to application and resource dynamics and user-friendly programming interface for providing application modelling and migrating facilities (El Kabbany *et al.*, 2011; Elmroth and Larsson, 2009).

Considering the second research topic, both the programming model and language must be carefully analyzed for trading-off between performance and usability. In this way, BSP (Bulk Synchronous Parallel) model and Java language appear as candidates to balance both aspects for grid computing (Bonorden, 2007). BSP represents a common style for writing successful round-based parallel programs (De Grande and Boukerche, 2011; Hendrickson, 2009; Bonorden, 2007). Lattice Boltzmann, DNA sequencing and weather forecast are examples of problems implemented with this model. Meanwhile, Java has a multi-platform characteristic and offers classes and methods for distributed computing that hide technical details (communication establishment between pairs and Sockets management, for instance) from the developers. Since this language is interpreted, it has

received much attention from the scientific community in order to turn its performance comparable with the execution time of binary codes. JIT (Just In Time), JNI (Java Native Interface), as well as improvements on Java Threads, garbage collector and memory management are examples of some initiatives to make Java faster and feasible for parallel computing (Taboada *et al.*, 2009).

Considering this context, we are developing a programming library called jMigBSP. It was designed to act over BSP-based Java applications and its differential approach concerns the offering of the rescheduling facility in different ways: at middleware and application levels. jMigBSP takes profit from the ProActive library for implementing object migration and to work over multiple-clusters based grids (Baduel *et al.*, 2006). Besides, our library aims to provide flexibility by providing one-sided asynchronous-typed communication among the objects.

Finally, we developed three BSP applications in order to validate our proposal: (*i*) Prefix Sum; (*ii*) Fractal Image Compression (FIC) and; (*iii*) Fast Fourier Transform (FFT). They show jMigBSP as viable solution to offer load balancing on BSP applications. Especially, FFT was also developed with the most used C-based BSP library, denoted BSPlib (Hill *et al.*, 1998). Besides a jMigBSP's description, the results of the article showed that the higher the computation grain, the lower the overhead imposed by an interpreted language. Furthermore, the migration tests revealed situations where jMigBSP outperforms BSPlib and emphasized the benefits of using this facility.

This article is organized in six sections. After the introduction section, we present the library proposed in this work. Thus, Section 2 is the most important part of the document. Section 3 shows the evaluation of the developed library. Section 4 presents the state-of-the-art libraries to write BSP applications. Finally, we show Section 5 at the end of this document. It makes a conclusion about the text, emphasizing the jMigBSP's main contributions and results.

jMigBSP: Java-Based BSP Communication Library for Object Rescheduling

jMigBSP library offers a Java interface (API) to write round-based applications like BSP. Basically, it inherits pertinent features from ProActive and proposes some modifications and new mechanisms to follow the BSP model strictly. jMigBSP takes profit from ProActive in four aspects: (*i*) resource deployment; (*ii*) Object-Oriented SPMD (OOSPMD) programming model; (*iii*) object migration; (*iv*) asynchronous communication. The former aspect allows informing the grid topology and the first object-node deployment. Both descriptions are presented in XML files, avoiding the specification of machine names and communication protocols in the application code directly. In addition, this approach makes the execution on different configurations possible without changing the application.

The OOSPMD paradigm launches multiple objects from the same class in different nodes and creates a specific communication group among them. Each object starts its execution in a determined method and the interaction among them happens through method invocation. The code of a method on the sender object performs an invocation of a remote method on the receiver object, characterizing an interaction known as one-sided. In addition, ProActive offers migration by leaving a proxy on the source node in which is used for performing calls to the migrated object. Finally, the asynchrony capability in jMigBSP is expressed by ProActive's mechanisms known as wait-by-necessity and Future Objects. A RMI always returns a Future Object to the caller, enabling it to continue its computation immediately. This object is changed by the real response on the fly (transparently to the caller) or the caller stops its execution when an enquiry is done over the Future until the answer comes through the network.

The first difference between jMigBSP and ProActive comprises the programming interface. Instead of offering Proactive-like API to the user, jMigBSP hides all complexity to understand ProActive interface by providing six methods. For example, some details about XML descriptors loading and group creation and management are abstracted from developers through a single method in jMigBSP. An application with jMigBSP must implement the semantic of a superstep in the *run()* method (see Section 2.1). A superstep presents both computation and communication actions followed by a synchronization barrier. A collection of them assembles a BSP application. As BSP does, jMigBSP also ensures that a message passed in a specific su-

perstep should be delivery only in the beginning of the next one.

Developing Applications with jMigBSP

A program with jMigBSP may be written in a single class. For that, any parallel algorithm extends jMigBSP class and implements the *run()* method. This method should contain the code that will execute concurrently. Table 1 shows the functions that can be used in the *run()* method. Firstly, the number of concurrent objects can be queried by calling *bsp_nprocs()*, and a unique object identifier can be retrieved through using *bsp_pid()*. *bsp_sync()* is a barrier that synchronizes all objects. When finishing this function, all messages sent in the previous superstep are available for use in the buffer on the destination object.

jMigBSP defines one-sided communication operation which allows users to read from or write to the memory of a remote process directly. Thus, the local buffer of each process can be manipulated by other processes by taking either *bsp_put()* or *bsp_get()*. *bsp_put()* stores data into the memory of a target process, without the active participation of this last entity. In the implementation aspect, all processes have two vectors of buffering in order to implement the BSP communication semantic. One vector has the data that should be used for sending data during a superstep, while the other is complete with receiving data. The former vector is called Active while the second is Temporary. Moreover, both vectors have size n, where n is the number of processes. After finalizing a superstep, the Active vector is filled by the content presented in the Temporary one. The operation *bsp_get()* reaches the local buffer of another process in order to copy data values held there into a data structure in its own local memory.

A shorthand way of using jMigBSP functions is illustrated in Figure 1. It explains the Prefix Sum computation operation where p sequential integers are stored on p processors. The algorithm uses the logarithmic technique in which performs log(p) supersteps. Considering this, the processes in the range $2^{k-1} \leq i \leq p$ combine their partial sums during the k^{th} superstep. Since a jMigBSP program is in essence a class, it is necessary to create its instance in order to run a program. However, creating an instance of a class does not implic-

Table 1. Collection of jMigBSP's methods and their classifications

Operation	Meaning
bsp_nprocs()	Number of objects
bsp_pid()	Find my identifier
bsp_sync()	Barrier synchronyzation
bsp_put()	Copy to remote memory
bsp_get()	Copy from remote memory
bsp_migrate()	Migrate the caller object

```
1.  public class PrefixSum extends jMigBSP {
2.    public void run() {
3.      int n = bsp_pid() + 1;
4.      for (int i = 1; i <bsp_nprocs(); i *= 2){
5.        if (bsp_pid() + i <bsp_nprocs())
6.          bsp_put((Object) n, bsp_pid() + i);
7.        bsp_sync();
8.        n = n + getBuffer(bsp_pid() - i);
9.      }
10.   }
11.   public static void main(String[] args) {
12.     PrefixSum s = new PrefixSum();
13.     s.start(4);
14.   }
15. }
```

Figure 1. Prefix Sum code written with jMigBSP.

itly start it. A call to *run(int p)* method must be performed to start the parallel phase using the given number of processes *p*, as shown in line 13 of Figure 1. It is important to observe that the mentioned program does not use *bsp_get()*. One-sided communication enables this feature, where data transferred in a superstep can be automatically captured in the beginning of the next one.

Figure 2 shows the supersteps involved in the Prefix Sum computation when working with 4 processors. Each process begins the algorithm with an integer that means its own identifier. In the first superstep, each process *i* sends a value to its neighbour on the right *i* + 1 (line 6 of Figure 1). The last process does not do that, since it does not have a neighbour on this position. The sending process will place its value into a Temporary buffer in the destination process. The barrier phase will complete the Active vector as explained earlier. Each process (except the first one) will now contain a value that is a sum of two integers. The resulting state of the first superstep is shown in Figure 2 on the second row. Following the algorithm, the last process will have the correct result in the third superstep.

Explicit Object Rescheduling Through Migration Calls

Object rescheduling allows the object-nodes remapping in response to application and infrastructure behaviour. jMigBSP library provides a way to migrate any BSP process from any JVM to any other one through calling *bsp_migrate()* explicitly. The migration can be initiated by the process itself or by an external agent. jMigBSP offers two implementation of *bsp_migrate()*. The first one consists in migrating the caller object to a remote host, which is received as input parameter. In order to do that, the remote host must have running a Java object called Node, a kind of daemon of ProActive library in charge of receiving and restarting active objects as well as keeping trace of locally accessible active objects. The other way to migrate considers the transferring of the caller process to a remote host in which another object executes currently. This *bsp_migrate()* signature has an object as input parameter.

In terms of BSP programs, a trivial way for rescheduling launching is to put migration directives after the barrier. This point represents a consistent global state, causing migration implementation and the capture of scheduling data easier. This last sentence is argued by the fact that the start of a superstep enables the decision making by using a global knowledge. In other words, updated data from all processes and nodes can be used in the synchronization operation for decision making on replacement (Kwok and Cheung, 2004).

The explicit rescheduling requires a developer with expertise in load balancing algorithms. In this way, he/she must collect data about processors' capacity and load for decision making on process relocation manually. Furthermore, the explicit rescheduling takes time away from the developers' primary interest: the application. Finally, a new application or/and new infrastructure of resources requires a new effort on studying the better places to add migration calls, as well as to choose pertinent supersteps for that.

Automatic Load Balancing

Besides explicit migrations, other jMigBSP's objective consists in providing automatic

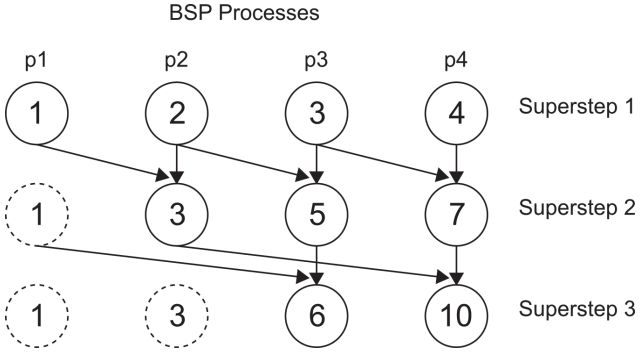

Figure 2. Prefix Sum operation using the logarithmic technique.

load balancing without developers/administrators intervention. In this way, MigBSP rescheduling model will be implemented and offered through a jMigBSP interface (Righi *et al.*, 2010). MigBSP answers the following issues: (*i*) "When" to launch the migration; (*ii*) "Which" objects are candidates for migration; (*iii*) "Where" to put an elected object. The article presented in (Righi *et al.*, 2010) describes the ideas to deal with these questions in detail. MigBSP runs over an architecture that is assembled with Sets (different sites or clusters) and Set Managers. Set Managers are responsible for scheduling, capturing data from a specific Set and exchanging it among other managers. The decision for automatic object remapping is taken at the end of a superstep. The term automatic means that the user/programmer will not put explicitly calls for processes rescheduling inside her/his application. Figure 3(a) illustrates a situation where the application code is changed and calls for rescheduling are inserted in the processes *p1* and *p5*. On the other hand, Figure 3(b) shows the main idea of MigBSP, where changes in the application code are not required and the load balancing is offered in a transparent way. Aiming to generate the least intrusiveness in application as possible, it is applied adaptations that control the interval between supersteps. The basic idea is to enlarge this index if the objects are balanced, or to reduce it otherwise.

We will offer an implementation that reacts against application and resource dynamics by migrating objects between different nodes. Our final aim is to reduce the application time by making the supersteps shorter. MigBSP answers the "Which" question by using a decision function called Potential of Migration (*PM*). Each object *i* computes *n* functions *PM*(*i*, *j*), where *n* is the number of Sets and *j* means a Set. *PM*(*i*, *j*) is found using Computation, Communication and Memory metrics as follows: *PM*(*i*, *j*) = *Comp*(*i*, *j*) + *Comm*(*i*, *j*) - *Mem*(*i*, *j*). Computation and Communication act in favour of migration, while Memory works in an opposite direction.

We are developing a prototype for automatic load balancing by using MigBSP ideas. For that, we created a new class denoted LBjMigBSP which extends jMigBSP in order to add and capture scheduling data. Figure 4 illustrates the methods in which LBjMigBSP overwrites from jMigBSP. Both lines 2 and

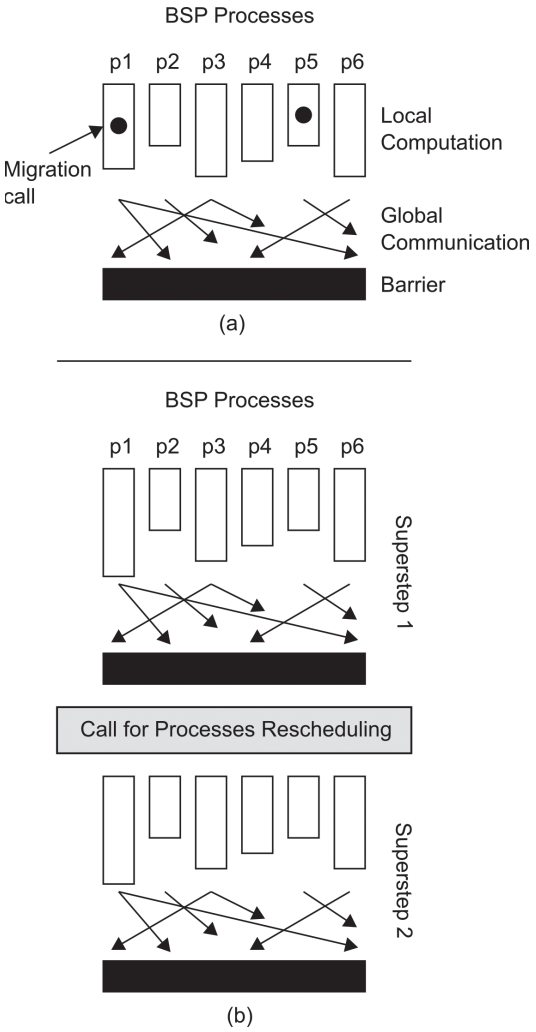

Figure 3. Two jMigBSP´s approaches for load balancing: (a) Changing the application code to offer explicit processes migration; (b) Processes rescheduling at middleware level without changing the application code.

6 from this figure show the communication methods. They capture scheduling information and trigger the jMigBSP method for treating message passing. *bsp_sync()* creates a vector to store the superstep time of each object. When rescheduling is activated, the code of *computeBalance()* method exchanges the vector among the Set Managers, allowing them to recognize the next interval of supersteps for object rescheduling. After achieving *PM*, the migration viability of each object is tested. The migrate method returns a new destination machine if migration is feasible, or null otherwis.

The developer must change only the line that defines the application class; by changing jMigBSP to LBjMigBSP (see line 1 of Figure 1).

```
1.  public class LBjMigBSP extends jMigBSP {
2.    public void bsp_put(Object o, int destination) {
3.    captureDataPut(o, destination);
4.    super.bsp_put(o,destination);
5.    }
6.    public Object bsp_get(int destination) {
7.      Object temp = super.bsp_get(destination);
8.     captureDataReceive(temp, destination);
9.      return temp;
10.  }
11.   public void bsp_sync(){
12.     double[] vec_steps = new double[MAX_SUPERSTEPS];
13.     computeSuperstepTime(vec_steps);
14.     if (isSuperstepRescheduling()) {
15.       next_call = computeBalance(vec_steps);
16.       exchageDataAmongSetManagers( computePM() );
17.       machine = willMigrate();
18.       if (machine != null)
19.       bsp_migrate(machine);
20.     }
21.     super.bsp_sync();
22.   }
23. }
```

Figure 4. Deriving jMigBSP in order to offer automatic load balancing.

Thus, LBjMigBSP represents our middleware approach, acting as a wrapper for automatic load balancing. LBjMigBSP is an ongoing work. Computation metric from *PM* considers the instructions performed between barriers as well as the time spent on computation actions inside a superstep. Communication metrics works with the number of bytes sent or received to/from a specific Set and the bandwidth to reach it. Finally, Memory metric takes into consideration the object memory, the bandwidth to achieve a Set and the costs related to the migration tool. Concerning this last item, we will measure the overhead on migration operations when transferring an object of 1 byte.

Experimental Results

We implemented three BSP applications: (*i*) Prefix Sum; (*ii*) Fractal Image Compression (FIC) and; (*iii*) Fast Fourier Transform (FFT). These applications were executed in a cluster with 16 nodes Intel Core 2 Duo 2.93GHz. They are connected by 10 Mbps links. In addition, we employed ProActive 5.0.3 and BSPlib 1.4 with TCP/IP-based message passing.

Analyzing the Migration Costs

These tests perform a comparison between the times of migrating objects with ProActive and the times of transferring objects with Java Sockets simply. Three nodes were reserved for the tests. The first one runs a synthetic application that creates an Active Object in the second node with a specific amount of data. The application calls the migration directive over the Active Object, moving it to the third node. Besides this application, another was written to create a Java object and to transfer it using Sockets. It uses only two nodes. Figure 5 shows the migration time of objects using ProActive compared to the transfer time with Java Sockets.

Data from 1KB up to 32MB are allocated by the objects. The application was tested 10 times for each number of bytes and an arithmetic average was computed. Basically, both migration and transfer times have a linear behaviour. For instance, 8.49s was achieved when migrating an object with 10MB with ProActive, while 7.29s is found using Java Sockets. When allocating 32MB, 30.32s and 28.89s were observed with ProActive and Java Sockets, respectively. Thus, 1.20s and 1.43s are measured in both cases when subtracting the Java time from the ProActive one. In other words, the overhead imposed by ProActive's object migration does not depend on the size of manipulated data directly.

Verifying the jMigBSP's Functionality

Prefix Sum was implemented and executed in our cluster to validate the correct

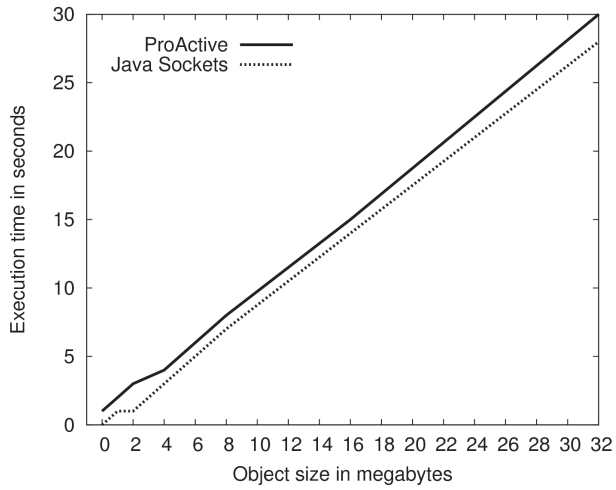

Figure 5. Observing the migration costs with ProActive.

implementation of jMigBSP. Our goal was to analyze the correct exchange of messages between objects and the synchronization between them during and after the barrier. The algorithm was executed with 4, 8 and 16 processes. These tests comprise the computation of 2, 3 and 4 supersteps, respectively. All tests resulted in the successfully application execution.

jMigBSP's Efficiency on Fractal-based Compression Application

FIC applications apply transformations which approximate smaller parts of the image by larger ones (LU, 1997). The smaller parts are called ranges and the larger ones domains. All ranges together form the image. The domains can be selected freely within the image. The application time increases as the number of domains increase as well. Our BSP modelling considers the variation of domain sizes as well as the number of objects.

The algorithm is based on circular pipeline configuration. First of all, each process receives its own range of domains. In this approach, each range block is compared with a different domain block at any given step. This comparison creates an index for each range, showing the transformation result when taking the domains of a specific process. Once the comparison step is completed the range blocks are shifted to the next processor in the pipeline, for another comparison step. When all range blocks have passed through the pipeline (which is actually a ring) the comparison step is completed for all domain blocks.

Table 2 presents the execution time of sequential FIC for an image with 256 *x* 256 pixels when varying the number of domains. Figure 6(a) show the image used in our evaluation. Figure 6(b), (c) and (d) show the resultant image using the number of domains of Scenarios 1, 2 and 3, respectively. In special, Figure 6(b) has few losses over the original image, achieving a compression rate of 68.52%. In addition, we can see that image compression increase while the number of domains is reduced. On the other hand, the greater the number of domains, the higher the quality of the resulting image and the lower is it compression rate.

Figure 7 shows the execution gains of FIC written with jMigBSP. We can observe that 8.15s was achieved in Scenario 1 with 16 processes. This result represents a gain of 92.92% over the sequential application. The execution time of Scenario 2 with 16 processes was 7.35s, while Scenario 3 with the same number of processes resulted in 5.97s. These results represent a gain of 92.91% and 92.90% when compared with the sequential application of Scenarios 2 and 3, respectively.

Comparing jMigBSP and BSPlib Native Library

We developed two versions of the FFT application: (*i*) one written in jMigBSP; and (*ii*) other in BSPlib. FFT is a fast algorithm for the computation of the Discrete Fourier Transform (DFT) and its inverse. Our implementation maps a complex-valued input vector of length to its DFT during the first two supersteps (Bisseling, 2004). Basically, the input

Table 2. Sequential execution time in seconds of Fractal Image Compression algorithm.

Scenario	Number of domains	Sequential execution time	Standard deviation	Compression rate
Scenario 1	4096	115.33	0.58	68.52%
Scenario 2	1024	103.78	1.75	91.93%
Scenario 3	256	84.21	1.17	97.76%

Figure 6. (a) Original image; (b) Resulting image with Scenario 1; (c) Resulting image with Scenario 2; (d) Resulting image with Scenario 3.

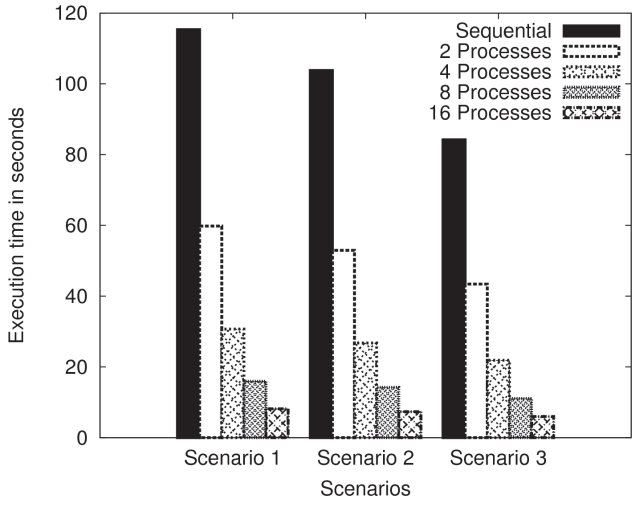

Figure 7. Observing the gains of parallel Fractal Image Compression algorithm with Scenarios 1, 2 and 3.

vector is assumed to be distributed cyclically over the processors. The algorithm starts with a parallel bit reversion, followed by a number of concurrent unordered sequential FFTs. Finally, it redistributes data and synchronizes the objects. The second superstep then proceeds with concurrent unordered sequential generalized FFTs. Both the size of the input vector and the number of processors are required to be powers of two.

Table 3 shows the execution times of FFT application. The vector size varied from 2^{23} to 2^{25}. As expected, BSPlib had a better performance against jMigBSP. This result is justified by the overhead imposed by Java and ProActive in jMigBSP. However, we can observe that the higher the computation grains the lower the difference time between both libraries. For instance, 7.13s was achieved with jMigBSP when using 16 processors and 2^{24} for vector size, while 5.33s was observed for BSPlib. These times represent an overhead of 25.24%. Nevertheless, vectors of 2^{25}, 16.25s and 14.41s were measured with jMigBSP and BSPlib, respectively. This execution informs a reduction of the jMigBSP's overhead, estimated in 23.63%.

Our evaluation also comprise observe the gains with processes migration. This test aims to validate the feasibility of developing jMigBSP's extension to offer automatic objects rescheduling. For this purpose, we executed FFT with vector size 2^{25} in 8 nodes and an overhead was simulated over 4 of them. To simulate overhead, we limit the maximum percentage of share of processor a work process receives when running using cpulimit[1] program (Cesario *et al.*, 2011; Vera and Suppi, 2011). We define 10% as the maximum of the processor that the FFT application can use in the 4 nodes. The application time written in BSPlib in this scenario was 61.76s, while the time of jMigBSP was 90.53s. These values represent an increase of 319.87% and 381.07% in the parallel application time written in BSPlib and jMigBSP in a dedicated environment, respectively.

Explicit calls to *bsp_migrate()* after the barrier were used in the tests. A migration of an object from an overloaded processor to a more lightly one leads in minimizing its execution time. We observed that 56.87s were obtained when migrated 4 processes located in overloaded nodes to others lightly loaded. This time represents gains in order of 37.17%, outperforming BSPlib execution time. The migration of 2 objects does not present satisfactory results due to the own BSP rule. The execution time achieved with this scenario was 88.68s. Despite transferring 2 objects, other 2 remain on overloaded processors and limit the superstep time.

Related Work

The state of the art on BSP programming comprises different libraries and languages. BSP libraries provide a full control over communication and synchronization, enabling users to write supersteps with different requirements easily. The Oxford BSPlib was one of the first BSP libraries (Hill *et al.*, 1998). It consists in implementations of the BSPlib standard in C, C++ and Fortran for multiple parallel architectures. Basically, it contains functions for delimiting supersteps and provides remote memory access (DRMA) and message passing (BSMP) operations. Other BSP libraries written in C language are the Paderborn University BSP (PUB) (Bonorden, 2007) and the BSPonMPI (Suijlen and Bisseling, 2011). PUB offers as the same functionality as Oxford BSPlib, but includes primitives for non-blocking communication and different semantics for barrier synchronization. BSPonMPI offers the basic functionalities from Oxford BSPlib and runs over MPI implementations.

Besides libraries in native code, we can describe some interpreted initiatives developed in Java. In this context, JBSP (Gu *et al.*, 2001), MulticoreBSP (Yzelman and Bisseling, 2011) and PUBWCL (Bonorden *et al.*, 2006) appear as the most significant ones. JBSP is a NOW-based system and provides both DRMA and BSMP communication methods. On the other hand, MulticoreBSP was designed for multicore systems where threads interact among themselves through a common shared memory. Especially, both PUBWCL and PUB have extensions to offer migration. PUBWCL library aims to take profit of idle cycles from nodes around the Internet. PUBWCL can migrate a BSP process during its computation phase, as well as after the barrier synchronization. The PUB's author proposed a centralized and distributed strategies for load balancing. In the first one, all nodes send data about their

[1] http://cpulimit.sourceforge.net/

CPU power and load to a master node. The master verifies the least and the most loaded node and migrates one process between them. In distributed approach, every node chooses *c* other nodes randomly and asks them for their load. One process is migrated if the minimum load of *c* analyzed nodes is smaller than own load of the node that is performing the test. Both PUBWCL and PUB just offer the migration approach with implicit migration calls.

Conclusion

This article presented the rationales for developing jMigBSP programming library. Its differential approach consists in providing BSP programming interface for Java with object migration facility. The migration can be triggered by the developer or not, depending on the degree of control desired. Implicit migration creates another class named LBjMigBSP in which extends jMigBSP in order to save scheduling data. LBjMigBSP implements a migration model that computes *PM* (Potential of Migration) of each object by considering both data of a computation and communication parts of a supersteps, as well as the migration costs.

Experimental evaluation showed encouraging results: (*i*) jMigBSP has a competitive performance if compared with the C-based library called BSPlib; (*ii*) migrations are pertinent for getting performance. Following this last statement, we concluded that highly-loaded applications tend to take more benefits from the migration facility. Future works comprises the development of LBjMigBSP and its evaluation with a CPU-Bound application. In addition, we intend to test jMigBSP on the Grid5000 (Cankar and Trobec, 2010) platform in order to observe its scalability, as well as the behaviour of the library when considering a real heterogeneous grid environment.

References

BADUEL, L.; FRANÇOISE, B.; CAROMEL, D.; CONTES, A.; HUET, F.; MOREL, M.; QUILICI, R. 2006. Programming, Composing, Deploying for the Grid. *In:* J.C. CUNHA; O.F. RANA, *Grid Computing: Software Environments and Tools*, Springer London, p. 205-229. http://dx.doi.org/10.1007/1-84628-339-6_9

BISSELING, R.H. 2004. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford, Oxford University Press, 334 p.

BONORDEN, O. 2007. Load Balancing in the Bulk-Synchronous-Parallel Setting using Process Migrations. *In:* IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS), 21, Long Beach, 2007. *Proceedings...* Long Beach, p. 1-9. http://dx.doi.org/10.1109/IPDPS.2007.370330

BONORDEN, O.; GEHWEILER, J.; HEIDE, F.M. 2006. Load Balancing Strategies in a Web Computing Environment. *In:* R. WYRZYKOWSKI; J. DONGARRA; N. MEYER; J. WASNIEWSKI, *Parallel Processing and Applied Mathematics*, Berlin, Springer, p. 839-846. http://dx.doi.org/10.1007/11752578_101

CANKAR, M.; TROBEC, R. 2010. Experimental evaluation of Grid5000 performance in the solution of PDE. *In:* MIPRO, 33, Ljubljana, 2010. *Proceedings...* Ljubljana, p. 203-207.

CESARIO, E.; GRILLO, A.; MASTROIANNI, C.; TALIA, D. 2011. A Sketch-Based Architecture for Mining Frequent Items and Itemsets from Distributed Data Streams. *In:* IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING (CCGrid), 11, Newport Beach, 2011. *Proceedings...* Newport Beach, p. 245-253. http://dx.doi.org/10.1109/CCGrid.2011.45

DE GRANDE, R.E.; BOUKERCHE, A. 2011. Dynamic balancing of communication and computation load for HLA-based simulations on large-scale distributed systems. *Journal of Parallel and Distributed Computing*, **71**(1):40-52. http://dx.doi.org/10.1016/j.jpdc.2010.04.001

EL KABBANY, G.F.; WANAS, N.; HEGAZI, N.; SHAHEEN, S. 2011. A Dynamic Load Balancing Framework for Real-time Applications in Message Passing Systems. *International Journal of Parallel Programming*, **39**(1):143-182. http://dx.doi.org/10.1007/s10766-010-0134-5

ELMROTH, E.; LARSSON, L. 2009. Interfaces for Placement, Migration, and Monitoring of Virtual Machines in Federated Clouds. *In:* INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING (GCC), 8, Lanzhou, 2009. *Proceedings...* Lanzhou, p. 253-260. http://dx.doi.org/10.1109/GCC.2009.36

GU, Y.; LEE, B.; CAI, W. 2001. JBSP: A BSP Programming Library in Java. *Journal of Parallel and Distributed Computing*, **61**(8):1126-1142. http://dx.doi.org/10.1006/jpdc.2001.1735

HENDRICKSON, B. 2009. Computational science: Emerging opportunities and challenges. *Journal of Physics: Conference Series*, **180**(1):012013. http://dx.doi.org/10.1088/1742-6596/180/1/012013

HILL, J.M.D.; MCCOLL, B.; STEFANESCU, D.C.; GOUDREAU, M.W.; LANG, K.; RAO, S.B.; SUEL, T.; TSANTILAS, T.; BISSELIN, R.H. 1998. BSPlib: The BSP programming library. *International Journal of Parallel Programming*, **24**(14):1947-1980. http://dx.doi.org/10.1016/S0167-8191(98)00093-3

KWOK, Y.; CHEUNG, L. 2004. A new fuzzy-decision based load balancing system for distributed object computing. *Journal of Parallel and Distributed Computing*, **64**(2):238-253. http://dx.doi.org/10.1016/j.jpdc.2003.11.002

LU, N. 1997. *Fractal Imaging*, San Francisco, Academic Press, 432 p.

PONTELLI, E.; LE, H.V.; SON, T.C. 2010. An investigation in parallel execution of answer set programs on distributed memory platforms: Task sharing and dynamic scheduling. *Computer Languages, Systems and Structures*, **36**(2):158-202. http://dx.doi.org/10.1016/j.cl.2009.09.001

RIGHI, R.R.; PILLA, L.L.; MAILLARD, N.; CARISSIMI, A.; NAVAUX, P.O.A. 2010. Observing the Impact of Multiple Metrics and Runtime Adaptations on BSP Process Rescheduling. *Parallel Processing Letters*, **20**(2):123-144. http://dx.doi.org/10.1142/S0129626410000107

SUIJLEN, W.J.; BISSELING, R.H. 2011. BSPonMPI. Available at: http://bsponmpi.sf.net/. Accessed on: 01/12/2011.

TABOADA, G.L.; TOURIÑO, J.; DOALLO, R. 2009. Java for high performance computing: assessment of current research and practice. *In:* INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA (PPPJ), 7, Calgary, 2009. *Proceedings...* Calgary, p. 30-39. http://doi.acm.org/10.1145/1596655.1596661

VERA, G.; SUPPI, R. 2011. ATLS - A parallel loop scheduling scheme for dynamic environments. *Procedia Computer Science*, **1**(1):583-591. http://dx.doi.org/10.1016/j.procs.2010.04.062

YZELMAN, A.N.; BISSELING, R.H. 2011. An Object-Oriented BSP Library for Multicore Programming. Available at: http://www.staff.science.uu.nl/~yzelm101/publications/yzelman11pp.pdf. Accessed on: 01/12/2011.

Submitted on October 10, 2011.
Accepted on December 12, 2011.