Scalable Computing: Practice and Experience
Volume 7, Number 2, pp. 1–14. http://www.scpe.org

ISSN 1895-1767
© 2006 SWPS

A WEB COMPUTING ENVIRONMENT FOR PARALLEL ALGORITHMS IN JAVA

OLAF BONORDEN* , JOACHIM GEHWEILER* , AND FRIEDHELM MEYER AUF DER HEIDE*

Abstract. We present a web computing library (PUBWCL) in Java that allows to execute tightly coupled, massively parallel algorithms in the bulk-synchronous (BSP) style on PCs distributed over the internet whose owners are willing to donate their unused computation power. PUBWCL is realized as a peer-to-peer system and features migration and restoration of BSP processes executed on it. The use of Java guarantees a high level of security and makes PUBWCL platform-independent. In order to estimate the loss of efficiency inherent in such a Java-based system, we have compared it to our C-based PUB-Library.

As the unused computation power of the participating PCs is unpredictable, we need novel strategies for load balancing that have no access to future changes of the computation power available for the application. We develop, analyze, and compare different load balancing strategies for PUBWCL. In order to handle the influence of the fluctuating available computation power, we classify the external work load.

During our evaluation of the load balancing algorithms we simulated the external work load in order to have repeatable testing conditions. With the best performing load balancing strategy we could save 39% of the execution time on average and even up to 50% in particular cases, in our test environment.

Key words. Bulk-Synchronous Parallel (BSP) Model, Web Computing, Volunteer-Based Computing, Scheduling, Load Balancing, Fault Tolerance, Java, Thread Migration

1. Introduction.

Bearing in mind how many PCs do exist distributed all over the world, one can easily imagine that all their idle times together represent a huge amount of unused computation power. There are already several approaches geared to utilize this unused computation power, for example:

- *distributed.net* [3]
- *Great Internet Mersenne Prime Search (GIMPS)* [7]
- *Search for Extraterrestrial Intelligence (SETI@home)* [18]

A common characteristic of most of these approaches is that the computational problem to be solved has to be divided into many small subproblems by a central server; clients on all the participating PCs download a subproblem, solve it, send the results back to the server, and continue with the next subproblem. Since there is no direct communication between the clients, only independent subproblems can be solved by the clients in parallel.

Our contribution. We have developed a web computing library (PUBWCL) that removes this restriction; in particular, it allows to execute tightly coupled, massively parallel algorithms in the bulk-synchronous (BSP) style on PCs distributed over the internet. PUBWCL is written in Java to guarantee a high level of security and to be platform independent.

When utilizing the unused computation power in a web computing environment, one has to deal with unpredictable fluctuations of the available computation power on the particular computers. Especially in a set of tightly coupled parallel processes, one single process receiving little computation power can slow down the whole application. A way to balance the load is to migrate these "slow" processes. PUBWCL therefore features migration of the BSP processes executed on it. In particular, we have implemented and analyzed four different load balancing strategies. PUBWCL furthermore features restoration of the BSP processes in order to increase fault tolerance.

Related work. Like PUBWCL, *Oxford BSPlib* [8] and *Paderborn University BSP Library (PUB)* [2, 13] are systems to execute tightly coupled, massively parallel algorithms according to the BSP model (see Section 2). They are written in C and are available for several platforms. These BSP libraries are optimized for application on monolithic parallel computers and clusters of workstations. These systems have to be centrally administered, whereas PUBWCL runs on the internet, taking advantage of Java's security model and portability.

The *Bayanihan* BSP implementation [16] follows the master-worker-paradigm: The master decomposes the BSP program to be executed into pieces of work, each consisting of one superstep in one BSP process. The workers download a packet consisting of the process state and the incoming messages, execute the superstep, and send the resulting state together with the outgoing messages back to the master. When the master has received the results of the current superstep for all BSP processes, it moves the messages to their destination

[*] Heinz Nixdorf Institute, Computer Science Departement, Paderborn University, 33095 Paderborn, Germany, {bono, joge, fmadh}@uni-paderborn.de

packets. Then the workers continue with the next superstep. With this approach all communication between the BSP processes passes though the server, whereas the BSP processes communicate directly in PUBWCL.

In [11], the problem of scheduling BSP processes on idle times of the processors is formalized as an online problem. It is shown that in the worst case, the competitive ratio, i. e., the factor by which the BSP algorithm is executed slower in the online setting, compared to an optimal offline setting, is arbitrarily large. The main contribution are two models that restrict the way how the unused computation power of the system changes over time, and algorithms with very small competitive ratio for these models. Our formalization of external work load in Section 6 is inspired by these results.

Organization of paper. The rest of the paper is organized as follows: In Sections 2 and 3, we give an overview of the used parallel computing model and the Java thread migration mechanism. In Section 4, we describe our web computing library. In Sections 5 and 6, we describe the implemented load balancing strategies and analyze the external work load. In Section 7, we evaluate the performance of our Java-based implementation and discuss the results obtained from experiments with our load balancing strategies. Section 8 concludes this paper.

2. The BSP Model. In order to simplify the development of parallel algorithms, Leslie G. Valiant has introduced the *Bulk-Synchronous Parallel (BSP)* model [20] which forms a bridge between the hardware to use and the software to develop. It gives the developer an abstract view of the technical structure and the communication features of the hardware to use (e. g. a parallel computer, a cluster of workstations or a set of PCs interconnected by the internet).

A *BSP computer* is defined as a set of processors with local memory, interconnected by a communication mechanism (e. g. a network or shared memory) capable of point-to-point communication, and a barrier synchronization mechanism (cf. Fig. 2.1).

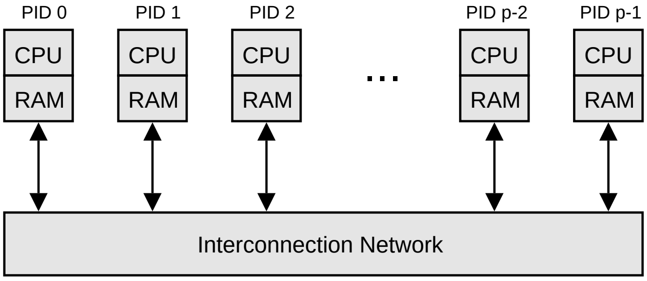

Fig. 2.1. *BSP computer*

A *BSP program* consists of a set of *BSP processes* and a sequence of *supersteps*—time intervals bounded by the barrier synchronization. Within a superstep each process performs local computations and sends messages to other processes; afterwards it indicates by calling the sync method that it is ready for the barrier synchronization. When all processes have invoked the sync method and all messages are delivered, the next superstep begins. Then the messages sent during the previous superstep can be accessed by its recipients. Fig. 3.1 illustrates this.

3. Thread Migration in Java. Complex algorithms often require much computation power, which means that they run for quite a long time, even on a parallel computer. Thus, we have the following problem in a web computing environment: If the owner of one of the PCs, whose unused computation power is donated, needs all his resources himself again, the BSP process running on that machine will take much more time to complete the current superstep. As shown in Fig. 3.1, this will delay the execution of the whole BSP program due to the barrier synchronization.

Thus, the execution time of a parallel program can be significantly improved if it is possible to migrate its processes at run-time to other hosts with currently more available computation power. Different load balancing strategies, that are used to decide when to migrate which BSP processes to which PUBWCL client, are presented in Section 5.

From the operating system's point of view, BSP processes are threads, so we actually need to migrate Java threads. There are three ways how this can be accomplished:

- modification of the Java Virtual Machine (VM) [12],
- bytecode transformations [15, 19],
- sourcecode transformations [17, 4].

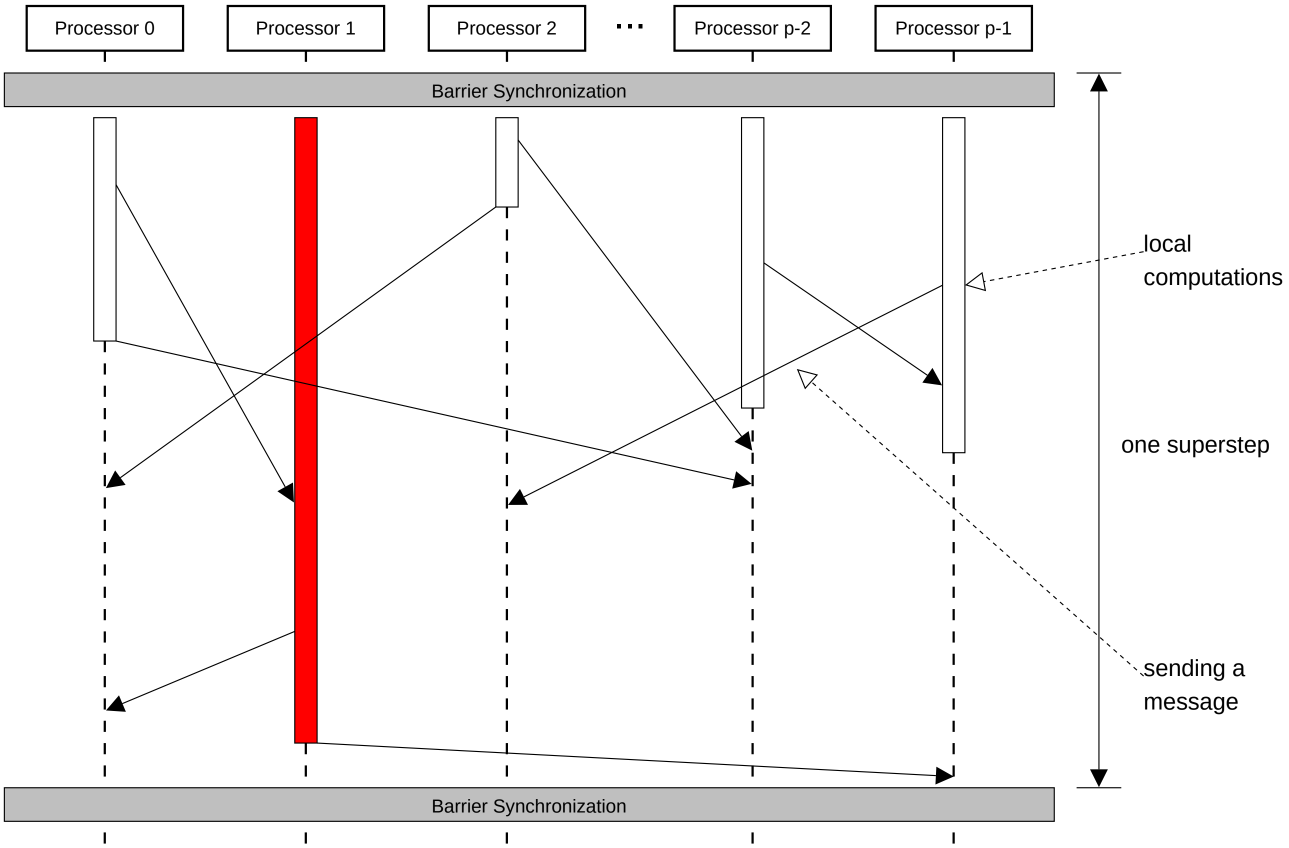

Fig. 3.1. *Delayed synchronization*

Modifying the Java VM is not advisable because everybody would have to replace his installation of the original Java VM with one from a third party, just to run a migratable Java program.

Inside PUBWCL, we use *JavaGo RMI* [17, 10] which is an implementation of the sourcecode transformation approach. It extends the Java programming language with three features:

- Migrations are performed using the keyword `go` (passing a filename instead of a hostname as parameter creates a backup copy of the execution state).
- All methods, inside which a migration may take place, have to be declared `migratory`.
- The depth, up to which the stack will be migrated, can be bounded using the `undock` statement.

The JavaGo compiler *jgoc* translates this extended language into Java sourcecode, using the unfolding technique described in [17]. Migratable programs are executed by dint of the wrapper `javago.Run`. In order to continue the execution of a migratable program, an instance of `javago.BasicServer` has to run on the destination host.

Since the original implementation of JavaGo is not fully compatible with the Java RMI standard, we use our own adapted version JavaGo RMI.

4. The Web Computing Library. People willing to join the *Paderborn University BSP-based Web Computing Library (PUBWCL)* system, have to install a PUBWCL client. With this client, they can donate their unused computation power and also run their own parallel programs.

Architecture of the system. PUBWCL is a hybrid peer-to-peer system: The execution of parallel programs is carried out on peer-to-peer basis, i. e., among the clients assigned to a task. Administrative tasks (e. g. user management) and the scheduling (i. e. assignment of clients and selection of appropriate migration targets), however, are performed on client-server-basis. Clients in private subnets connect to the PUBWCL system via the *proxy* component. The interaction of the components is illustrated in Fig. 4.1.

Since the clients may join or leave the PUBWCL system at any time, the login mechanism is lease-based, i. e., a login session expires after some timeout if the client does not regularly report back at the server.

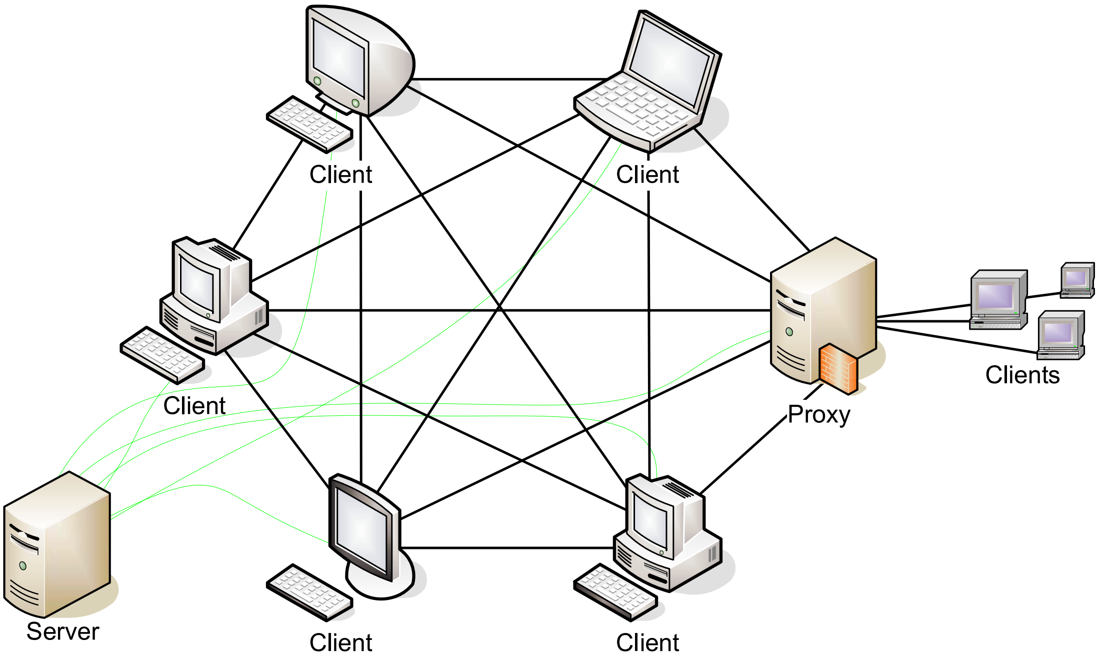

Fig. 4.1. *The architecture of PUBWCL.*

Though a permanent internet connection is required, changes of dynamically assigned IP addresses can be handled. This is accomplished by using *Global Unique Identifiers (GUIDs)* to unambiguously identify the clients: when logging in, each client is assigned a GUID by the server. This GUID can be resolved into the client's current IP address and port.

Executing a parallel program. If users want to execute their own parallel programs, they must be registered PUBWCL users (otherwise, if they only want to donate their unused computation power, it is sufficient to use the guest login). To run a BSP program, it simply has to be copied into a special directory specified in the configuration file. Then one just needs to enter the name of the program and the requested number of parallel processes into a dialog form. Optionally, one may pass command line arguments to the program or choose a certain load balancing algorithm.

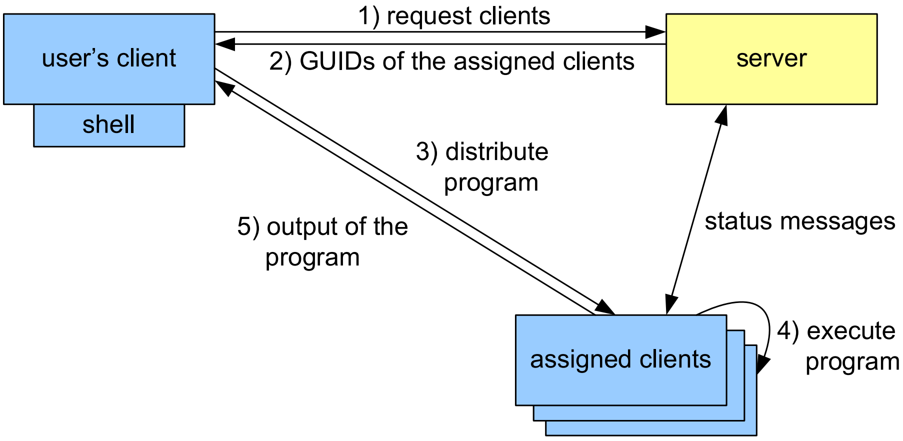

Fig. 4.2. *Executing a BSP program in PUBWCL.*

The server then assigns the parallel program to a set of clients and sends a list of these clients to the user's client (cf. Fig. 4.2). From now on, the execution of the parallel program is supervised by the user's client. On each of the assigned clients a PUBWCL runtime environment is started and the user's parallel program is obtained via dynamic code downloading. The output of the parallel program and, possibly, error messages including stack traces are forwarded to the user's client.

All processes of parallel programs are executed in an own PUBWCL runtime environment in a separate process, so that it is possible to cleanly abort single parallel processes (e. g. in case of an error in a user program).

Security aspects. The *Java Sandbox* allows to grant code specific permissions depending on its origin. For example, access to (part of) the file system or network can be denied. In order to guarantee a high level of security, we grant user programs only read access to a few Java properties which are needed to write completely platform independent code (e. g. `line.separator` etc.).

Details on the internals of the library as well as a guide how to configure it can be found in [14] and [5].

4.1. The Programming Interface. User programs intended to run on PUBWCL have to be BSP programs ([1] is an excellent guide to parallel scientific compuation using BSP). Thereto the interface `BSPProgram` must be implemented, i. e., the program must have a method with this signature:

```
public void bspMain(BSP bspLib, String[] args)
    throws AbortedException
```

Its first parameter is a reference to the PUBWCL runtime environment which supports the BSP interface; the second parameter is an array containing the command line parameters passed to the BSP program.

In order to write a migratable program, the interface `BSPMigratableProgram` has to be implemented instead, which means that the main method has this different signature:

```
public migratory void bspMain(BSPMigratable bspLib,
    String[] args) throws AbortedException, NotifyGone
```

The following BSP library functions can be accessed via the `BSP` resp. `BSPMigratable` interface which is implemented by the PUBWCL runtime environment:

In non-migratable programs, the barrier synchronization is entered by calling:

```
public void sync()
```

The migratable version additionally creates backup copies of the execution state and performs migrations if suggested by the load balancing strategy:

```
public migratory void syncMig() throws NotifyGone
```

A message, wich can be any serializable Java object, can be sent with these methods; thereby the latter two methods are for broadcasting a message to an interval resp. an arbitrary subset of the BSP processes:

```
public void send(int to, Serializable msg)
    throws IntegrityException
public void send(int pidLow, int pidHigh, Serializable msg)
    throws IntegrityException
public void send(int[] pids, Serializable msg)
    throws IntegrityException
```

Messages sent in the previous superstep can be accessed with these methods, where the `find*` methods are for accessing messages of a specific sender:

```
public int getNumberOfMessages()
public Message getMessage(int index)
    throws IntegrityException
public Message[] getAllMessages()
public Message findMessage(int src, int index)
    throws IntegrityException
public Message[] findAllMessages(int src)
    throws IntegrityException
```

When receiving a message, it is encapsulated in a `Message` object. The message itself as well as the sender ID can get obtained with these methods:

```
public Serializable getContent()
public int getSource()
```

In order to terminate all the processes of a BSP program, e. g. in case of an error, the following method has to be called; the `Throwable` parameter will be transmitted to the PUBWCL user who has started the program:

```
public void abort(Throwable cause)
```

Any output to *stdout* or *stderr* should be printed using the following methods as they display it on the PUBWCL client of the user who has started the program rather than on the computer where the process is actually running:

```
public void printStdOut(String line)
public void printStdErr(String line)
```

To access data from files, the following method should be used. In particular, any file in the BSP program folder of the user's client can be read with it:

```
public InputStream getResourceAsStream(String name)
    throws ChainedException
```

In migratable programs, there is also a method available which may be called to mark additional points inside long supersteps where a migration is safe (i. e. no open files etc.):

```
public migratory boolean mayMigrate() throws NotifyGone
```

Furthermore, there are some service functions to obtain the number of processes of the BSP program, the own process ID, and so on.

Listing 1 shows an example program which demonstrates the basic BSP features, especially how to send and receive messages.

Listing 1
Example program demonstrating message passing.

```
import de.upb.sfb376.a1.*;
import de.upb.sfb376.a1.bsp.*;

public class MessagePassing implements BSPProgram
{
  public void bspMain(BSP bsp, String[] args) {
    int i, left, right;
    Message msg;
    Message[] msgs;

    // calculate neighbours
    left = (bsp.getPID() + bsp.getNumberOfProcessors() - 1) % bsp.getNumberOfProcessors();
    right = (bsp.getPID() + 1) % bsp.getNumberOfProcessors();

    try {
      bsp.send(left, new Integer(1));
      bsp.send(right, new Integer(2));
    } catch(IntegrityException ie) {
      bsp.printStdErr("an error occurred during ’send’: ‘‘ + ie.getMessage());
    }
    bsp.sync();

    // get all messages, method 1
    for(i=0; i<bsp.getNumberOfMessages(); i++) {
      try {
        msg = bsp.getMessage(i);
        bsp.printStdOut("recieved ‘‘ + msg.getContent() + ‘‘ from pid ‘‘ + msg.getSource() + ‘‘ in superstep ‘‘ + msg.getSuperstep());
      } catch(IntegrityException ie) {
        bsp.printStdErr("an error occurred during ’getMessage’: ‘‘ + ie.getMessage());
      }
    }

    // get all messages, method 2
    msgs = bsp.getAllMessages();
    bsp.printStdOut("recieved in total ‘‘ + msgs.length + ‘‘ messages");

    // get messages from some specified pid, method 1
    try {
      i = 0;
      while((msg = bsp.findMessage(0, i++)) != null)
        bsp.printStdOut("recieved ‘‘ + msg.getContent() + ‘‘ from pid 0 in superstep ‘‘ + msg.getSuperstep());
    } catch(IntegrityException ie) {
      bsp.printStdErr("an error occurred during ’findMessage’: ‘‘ + ie.getMessage());
    }

    // get messages from some specified pid, method 2
    try {
      msgs = bsp.findAllMessages(0);
      bsp.printStdOut("recieved ‘‘ + msgs.length + ‘‘ messages from pid 0");
    } catch(IntegrityException ie) {
      bsp.printStdErr("an error occurred during ’findAllMessages’: ‘‘ + ie.getMessage());
    }
  }
}
```

5. Load Balancing. As already pointed out in Section 3, one single process receiving little computation power can slow down the execution of the whole BSP program due to the barrier synchronization. Migrating these "slow" BSP processes can therefore significantly improve the execution time of the BSP program. Before we present our load balancing algorithms, we need some preliminaries for scheduling.

First of all, we can derive the following constraint from the properties of a BSP algorithm. Since all the BSP processes are synchronized at the end of each superstep, we can reduce the scheduling problem for a BSP algorithm with n supersteps to n subproblems, namely scheduling within a superstep.

Second, we assume that we only have to deal with "good" BSP programs, i. e., all of the p BSP processes require approximately the same amount of computational work. Thus the scheduler has to assign p equally heavy pieces of work.

Finally, we have another restriction. Due to privacy reasons we cannot access the breakdown of the CPU usage, i. e., we especially do not know how much computation power is consumed by the user and how much computation power is currently assigned to our BSP processes.

Parameters for scheduling. Since we cannot directly access the breakdown of the CPU usage, we have to estimate (1) how much computation power the BSP processes currently assigned to a client do receive, and (2) how much computation power a BSP process would receive when (additionally) assigned to a client.

As from the second superstep on, the first question can simply be answered by the ratio of the computation time consumed during the previous superstep and the number of concurrently running BSP processes.

In order to answer the second question, all clients regularly measure the *Available Computation Power (ACP).* This value is defined as the computation power an additionally started BSP process would receive on a particular client, depending on the currently running processes and the external work load. We obtain this value by regularly starting a short benchmark process and measuring the computation power it receives. Though it is not possible to determine the CPU usage from the ACP value, this value is platform independent and thus comparable among all clients.

Since the first approach is more accurate, we will use it wherever possible, i. e., mainly, to decide whether a BSP process should migrate or has to be restarted. The ACP value will be used to determine the initial distribution of the BSP processes and to choose clients as migration targets and as hosts for restarted BSP processes.

5.1. The load balancing algorithms. We have implemented and analyzed the following four load balancing strategies, among them two parallel algorithms and two sequential ones.

Algorithm PwoR. The load balancing algorithm *Parallel Execution without Restarts (PwoR)* executes all processes of a given BSP program concurrently.

The initial distribution is determined by dint of the ACP values. Whenever a superstep is completed, all clients are checked whether the execution of the BSP processes on them took more than r times the average execution duration (a suitable value for r will be chosen in Section 7.2); in this case the BSP processes are redistributed among the active clients such that the expected execution duration for the next superstep is minimal, using as little migrations as possible.

Algorithm PwR. Using the the load balancing algorithm *Parallel Execution with Restarts (PwR)*, the execution of a superstep is performed in phases. The duration of a phase is r times the running time of the $\lceil s \cdot p^* \rceil$-th fastest of the (remaining) BSP processes, where p^* is the number of processes of the BSP program that have not yet completed the current superstep. Suitable values for the parameters $r > 1$ and $0 < s < 1$ will be chosen in Section 7.2. At the end of a phase, all incomplete BSP processes are aborted. In the next phase, they are restarted on faster clients. Whereas too slow BSP processes are migrated only after the end of the superstep using the PwoR algorithm, they are restarted already during the superstep using PwR.

At the end of each (except the last) superstep, the distribution of the BSP processes is optimized among the set of currently used clients by dint of the processes' execution times in the current superstep. The optimization of the distribution is performed such that the number of migrations is minimal.

Algorithm SwoJ. While the two load balancing strategies PwoR and PwR execute all BSP processes in parallel, the load balancing algorithm *Sequential Execution without Just-in-Time Assignments (SwoJ)* executes only one process of a BSP program per client at a time; the other BSP processes are kept in queues.

Like PwR, SwoJ operates in phases. At the end of a phase all uncompleted BSP processes are aborted and reassigned. Thereby the end of a phase is reached after r times the duration, in which the x-th fastest client has completed all assigned BSP processes, where x is a fraction s of the number of the affected clients. At the end of a superstep, the distribution of the BSP processes is optimized like in PwR.

Algorithm SwJ. Like SwoJ, the load balancing algorithm *Sequential Execution with Just-in-Time Assignments (SwJ)* executes only one process of a BSP program per client at a time and keeps the other processes in queues, too. The main difference, however, is that these queues are being balanced. More precisely, whenever a client has completed the execution of the last BSP process in its queue, a process is migrated to it from the queue of the most overloaded client (if there exists at least one overloaded client). Thereby a client is named "overloaded" in relation to another client if completing all but one BSP processes in the queue with the current execution speed of the particular client would take longer than executing one BSP process on the other client. Thereby the execution speed of a client is estimated by dint of the running time of the BSP process completed on it most recently.

BSP processes are aborted only if the corresponding queues on all affected clients are empty and if the processes do not complete within r times the duration of a process on the x-th fastest client, weighted by the number of BSP processes on the particular clients; thereby, again, x is a fraction s of the number of the affected clients.

6. The External Work Load. In order to understand the fluctuation of the external work load, we have analyzed totaling more than 100 PCs in altogether four departments of three German universities over a period of 21 resp. 28 days. The CPU frequencies varied from 233 MHz to 2.8 GHz. The installed operating systems were Debian Linux, RedHat Enterprise Linux, and SuSE Linux.

While analyzing the load, we have noticed that the CPU usage typically shows a continuous pattern for quite a time, then changes abruptly, then again shows a continuous pattern for some time, and so on. The reason therefore is that many users often perform uniform activities (e. g. word processing, programming, and so on) or no activity (e. g. at night or during lunch break).

A given CPU usage graph (e. g. of the length of a week) can thus be split into blocks, in which the CPU usage is somewhat steady or shows a continuous pattern. These blocks typically have a duration of some hours, but also durations from only half an hour (e. g. lunch break) up to several days (e. g. a weekend) do occur.

Based on the above observations we have designed a model to describe and classify the external work load. We describe the CPU usage in such a block by a rather tight interval with radius $\alpha \in \mathbb{R}$ ($\alpha < \frac{1}{2}$) around a median load value $\lambda \in \mathbb{R}$ ($0 \leq \lambda - \alpha,\ \lambda + \alpha \leq 1$), as illustrated in Fig. 6.1. The rates for the upper and lower deviations are bounded by $\beta^+ \in \mathbb{R}$ resp. $\beta^- \in \mathbb{R}$ ($\beta^+, \beta^- < \frac{1}{2}$). We will refer to such a block as a $(\lambda, \alpha, \beta^+, \beta^-, T)$-*load sequence* in the following.

In order to describe the frequency and duration of the deviations, we subdivide the load sequences into small sections of length T, called *load periods*. The values β^+ and β^- must be chosen such that the deviation rates never exceed them for an arbitrary starting point of a load period within the load sequence.

Let $D \in \mathbb{R}^+$ be the execution duration of a superstep of a BSP process in the case that we receive the full computation power of the used machine. Given that T is much shorter than the duration of a superstep, we can obtain this result:

THEOREM 6.1. *If a superstep of length D of a BSP process is executed completely within a $(\lambda, \alpha, \beta^+, \beta^-, T)$-load sequence, the factor between its minimal and maximal possible duration is at most $q' \in \mathbb{R}^+$ with*

$$q' \leq \delta(D) \cdot \frac{1 - (1 - \beta^-)(\lambda - \alpha)}{1 - (\beta^+ + (1 - \beta^+)(\lambda + \alpha))}$$

where $\delta(D)$ tends to 1 with $D \to \infty$. For $q \in \mathbb{R}^+$, $q \geq q'$ we call a $(\lambda, \alpha, \beta^+, \beta^-, T)$-load sequence q-bounded.

PROOF: Let $d(t) : \mathbb{R}_0^+ \mapsto \mathbb{R}^+$ denote the actually required execution time of the superstep depending on the external load if the execution starts at time $t \in \mathbb{R}_0^+$.

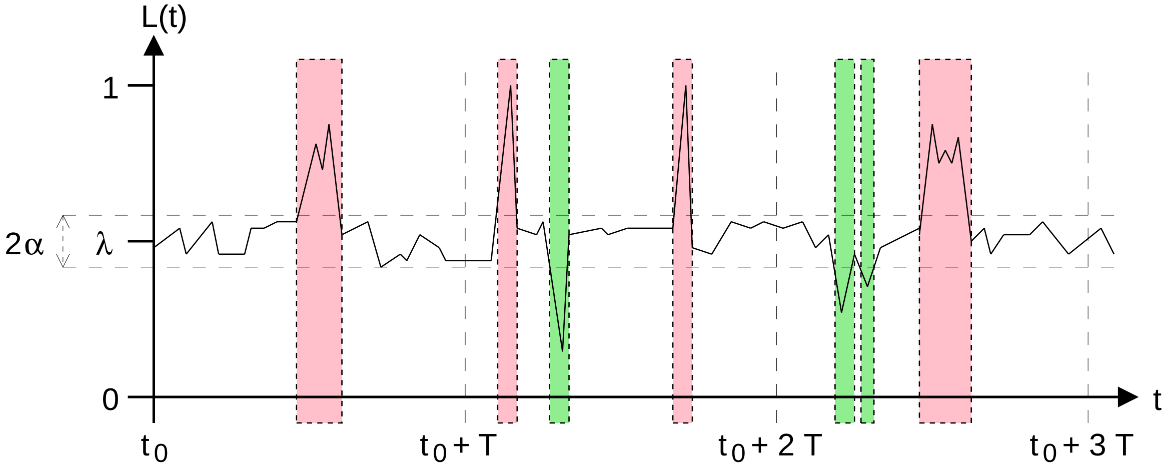

FIG. 6.1. *A three-load-period-long interval of a load sequence.*

First, we show that:

$$\frac{D}{1-(1-\beta^{-})(\lambda-\alpha)} - \varepsilon_1 \leq d(t) \leq \frac{D}{1-(\beta^{+}+(1-\beta^{+})(\lambda+\alpha))} + \varepsilon_2 \tag{6.1}$$

where:

$$\varepsilon_1 = \beta^{-}\, T\, \frac{\lambda-\alpha}{1-(\lambda-\alpha)} \text{ and } \varepsilon_2 = \beta^{+}\, T$$

As we are given that the superstep is executed completely within a $(\lambda, \alpha, \beta^{+}, \beta^{-}, T)$-load sequence, the external work load is bounded by:

$$\ell_{\text{min}} := \beta^{-} \cdot 0 + (1-\beta^{-})(\lambda-\alpha) \tag{6.2}$$

resp.

$$\ell_{\text{max}} := \beta^{+} \cdot 1 + (1-\beta^{+})(\lambda+\alpha) \tag{6.3}$$

Dividing the execution duration D by these bounds, we get (6.1). It remains to show that our estimations for the corrections values ε_1 and ε_2 hold. These values are needed because $d(t)$ is not necessarily a multiple of T.

Since a load period may begin at any point within a load sequence, we can assume w.l.o.g. that the execution of the superstep starts at the beginning of a load period. Thus, we only have a fractionally utilized load period at the end of the superstep.

Let $x \in \mathbb{R}$, $0 < x < 1$, be the fraction to which this load period is used. If there were no correction values, this would mean that at most a fraction x of the deviations in the load period would affect the execution of the superstep. But this is not necessarily the case as the deviations can occur at arbitrary points within the load period by definition. Thus, we have to choose ε_1 and ε_2 such that an arbitrary amount of the deviations can interfere with the execution of the superstep.

Fig. 6.2 shows the influences of the upper deviations (the gray area is the computation power that the BSP process receives): If all the upper deviations occur in the first part of the load period, the end of the execution is delayed by up to $\varepsilon_2 = \beta^{+}\, T$.[1]

In order to estimate the influences of the lower deviations, have a look at Fig. 6.3: The gray areas again are the computation power that the BSP process receives. The dark gray rectangle of case a) has been cut off in case b) and been replaced by a rectangle with the same area in the lower deviation. Thus, the execution duration

[1] This estimation actually is a bit too pessimistic as a fraction x of the deviations is already contained in (6.1) without the correction values.

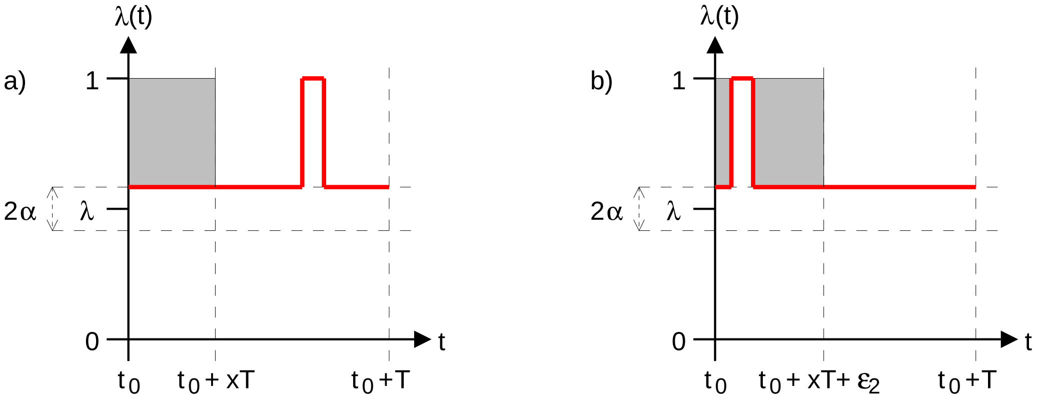

FIG. 6.2. *The influences of the upper deviations in the last load period.*

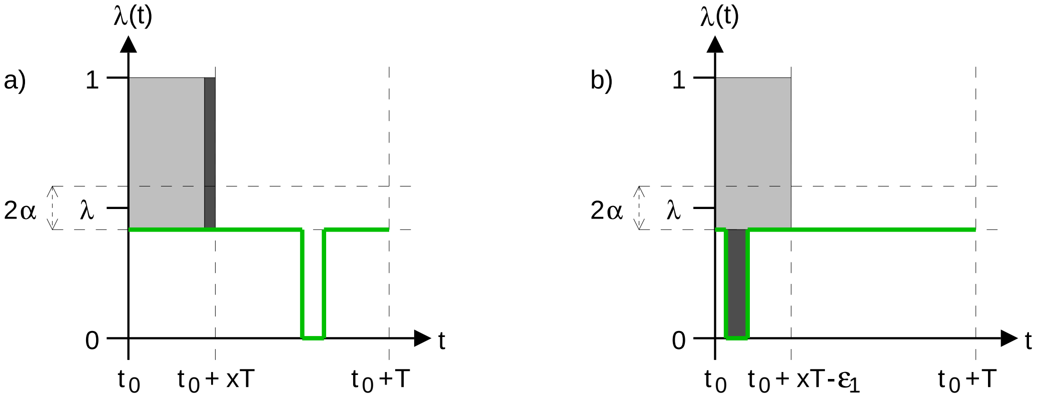

FIG. 6.3. *The influences of the lower deviations in the last load period.*

in case b) is shortened by the width of the dark gray rectangle of case a), which can be obtained by dividing the area of the dark gray rectangle of case b) by the height of the one of case a), i. e., $\varepsilon_1 = \beta^- T \frac{\lambda-\alpha}{1-(\lambda-\alpha)}$.[1]

Now we obtain the following estimation from (6.1) for some $q' \in \mathbb{R}^+$:

$$\frac{D}{1-(\beta^+ + (1-\beta^+)(\lambda+\alpha))} + \varepsilon_2 = q' \left(\frac{D}{1-(1-\beta^-)(\lambda-\alpha)} - \varepsilon_1 \right)$$

$$\Leftrightarrow \frac{D + \varepsilon_2(1-(\beta^+ + (1-\beta^+)(\lambda+\alpha)))}{1-(\beta^+ + (1-\beta^+)(\lambda+\alpha))} = q' \cdot \frac{D - \varepsilon_1(1-(1-\beta^-)(\lambda-\alpha))}{1-(1-\beta^-)(\lambda-\alpha)}$$

$$\Leftrightarrow q' = \frac{1-(1-\beta^-)(\lambda-\alpha)}{1-(\beta^+ + (1-\beta^+)(\lambda+\alpha))} \cdot \frac{D + \varepsilon_2(1-(\beta^+ + (1-\beta^+)(\lambda+\alpha)))}{D - \varepsilon_1(1-(1-\beta^-)(\lambda-\alpha))}$$

Now we define:

$$\delta(D) := \frac{D + \varepsilon_2(1-(\beta^+ + (1-\beta^+)(\lambda+\alpha)))}{D - \varepsilon_1(1-(1-\beta^-)(\lambda-\alpha))}$$

As we have $\varepsilon_1 = \beta^- T \frac{\lambda-\alpha}{1-(\lambda-\alpha)}$ and $\varepsilon_2 = \beta^+ T$ for some fixed T, we get:

$$\lim_{D\to\infty} \delta(D) \to 1$$

This concludes the proof. □

Theorem 6.1 guarantees that the running times of BSP processes, optimally scheduled based on the execution times of the previous superstep, differ at most by a factor q^2 within a load sequence. This fact will be utilized by the load balancing strategies.

Evaluating the collected data. When sectioning a given CPU usage sequence into load sequences, our goal is to obtain load sequences with a q-boundedness as small as possible and a duration as long as possible, while the rate of unusable time intervals should be as small as possible. Obviously, these three optimization targets depend on each other. We have processed the data collected from our PCs described in the beginning of this section (over 6.8 million samples) with a Perl program which yields an approximation for this non-trivial optimization problem.

The results. The average idle time over a week ranged from approx. 35% up to 95%, so there is obviously a huge amount of unused computation power. Time intervals of less than half an hour and such where the CPU is nearly fully utilized by the user or its usage fluctuates too heavily, are no candidates for a load sequence. The rate of wasted idle time in such intervals is less than 3%.

Choosing suitable values for the parameters of the load sequences, it was possible to section the given CPU usage sequences into load sequences such that the predominant part of the load sequences was 1.6-bounded.

On most PCs, the average duration of a load sequence was 4 hours or even much longer. Assuming the execution of a process, started at an arbitrary point during a load sequence, takes 30 minutes, the probability that it completes within the current load sequence is thus at least 87.5%. A detailed analysis of the results in each of the four networks can be found in [6].

Generating load profiles. In order to compare the load balancing strategies under the same circumstances, i. e., especially with exactly the same external work load, and to make experimental evaluations repeatable, we have extracted totaling eight typical load profiles from two of the networks, each using these time spans: Tuesday forenoon (9:00 a.m. to 1:00 p.m.), Tuesday afternoon (2:00 p.m. to 6:00 p.m.), Tuesday night (2:00 a.m. to 6:00 a.m.), and Sunday afternoon (2:00 p.m. to 6:00 p.m.). Besides, we have generated four artificial load profiles according to our model, using typical values for the parameters. A detailed discussion of the load profiles can be found in [6].

7. Experimental Evaluation. In the following we present a comparision of our Java-based library against a C-based implementation and the evaluation of our load balancing algorithms.

7.1. Performance Evaluation. In order to determine the performance drawback of PUBWCL in comparison to a BSP implementation in C, we have conducted benchmark tests with both PUBWCL and PUB under the same circumstances: We used a cluster of 48 dual Intel Pentium III Xeon 850 MHz machines, that were exclusively reserved for our experiments to avoid influences by external work load. The computers were interconnected by a switched Fast Ethernet. The used benchmark program was a sequence of 10 equal supersteps. Per superstep, each BSP process did a number of integer operations and sent a number of messages. We performed tests using every possible combination of these parameters:

- 8, 16, 24, 32, 48 BSP processes
- 10, 20, 30 messages per BSP process and superstep
- 10 kB, 50 kB, 100 kB message size
- 0, 10^8, $2 \cdot 10^8$, $3 \cdot 10^8$, ..., 10^9 integer operations per BSP process and superstep

Selected results of the benchmark tests are shown in Fig. 7.1. As you can see, both BSP libraries scale well, and there is a performance drawback of a factor 3.3. Note that the running time of this benchmark program is dominated by the sequential work. Communication tests with no sequential work showed a performance impact of a factor up to 8.7.

When porting existing BSP programs to PUBWCL, you cannot directly compare the running times due to the overhead of the Java memory management. For example, we ported our C-based solver for the 3-Satisfiability-Problem (3-SAT), which is a simple parallelized version of the sequential algorithm in [9], to PUBWCL. As in the case of the benchmark program, the algorithm is dominated by the sequential work. But in contrast to the benchmark program, it continuously creates, clones, and disposes complex Java objects. This is much slower than allocating, copying, and freeing structures in the corresponding C-program and has led to a performance drawback of a factor 5.4.

7.2. Evaluation of the Load Balancing Algorithms. In order to analyze our load balancing strategies, we have conducted experiments on 15 PCs running Windows XP Professional, among them 7 PCs with 933 MHz and 8 ones with 1.7 GHz. The PUBWCL server and the client used to control the experiments ran on a seperate PC.

We have simulated the external work load according to the load profiles mentioned in Section 6 and run

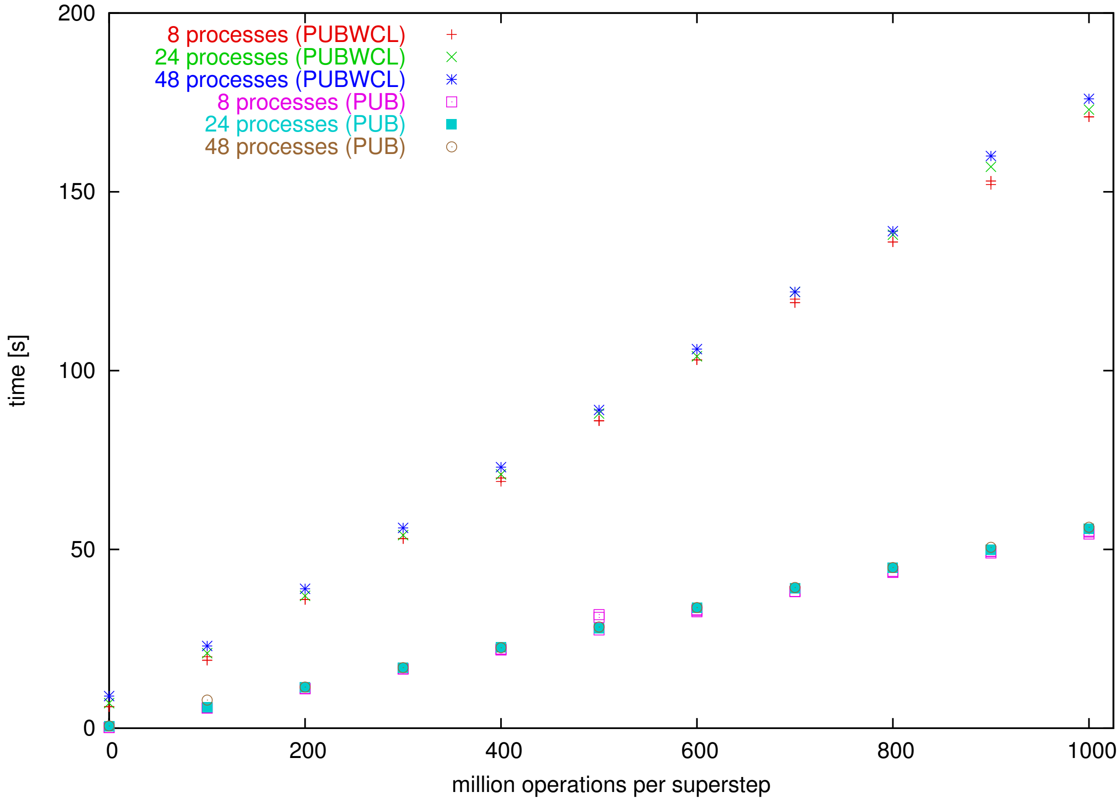

Fig. 7.1. *Results of the benchmark tests.*

the clients with the `/belownormal` priority switch, i. e., they could only consume the computation power left over by the load simulator.

The experiments were performed using a BSP benchmark program consisting of 80 equally weighted processes and 8 identical supersteps. Per superstep, each BSP process did $2 \cdot 10^9$ integer operations and sent (and received) 64 messages of 4 kB size each.

Results using PwoR. First we have to choose a suitable value for parameter r: At the beginning of a superstep the BSP processes are (re-) distributed over the clients such that they should complete execution at the same time. Supposing that, on any of the involved PCs, the current (q-bounded) load sequence does not end before completion of the superstep, none of the BSP processes should take longer than q times the average execution time. That means, choosing $r = q = 1.6$ guarantees that BSP processes are not migrated if the available computation power only varies within the scope of a q-bounded load sequence.

In comparison to experiments with no load balancing algorithm (i. e. initial distribution according to the ACP values and no redistribution of the processes during runtime), we could save 21% of the execution time averaged and even up to 36% in particular cases.

Comparing the execution times of the particular supersteps, we noticed that the execution time significantly decreases in the second superstep. The reason therefore is that the execution times of the previous superstep provide much more accurate values for load balancing than the estimated ACP values.

Results using PwR. Our experiments with the PwR algorithm resulted in noticealby longer execution times than those with the PwoR algorithm. We could obtain the best results with the parameters set to $r = 2$ and $s = \frac{1}{8}$; other choices led to even worse results.

On the one hand, this result is surprising as one would expect that PwR performs better than PwoR because it restarts BSP processes after some threshold instead of waiting for them for an arbitrarily long time. But on the other hand, restarting a BSP process is of no advantage if it would have completed on the original client within less time than its execution time on the new client. Our results show that the external work load apparently is not ‘sufficiently malicious’ for PwR to take advantage of its restart feature.

Results using SwoJ. Like with the PwR algorithm, $r = 2$ and $s = \frac{1}{8}$ is a good choice for the parameters because: In Section 6 we showed that a load sequence does not end inside a superstep with a probability of at least 87.5%. Thus the probability that a new load sequence with *more* available computation power starts inside a superstep is at most $\frac{1}{16}$. As we will use the normalized BSP process execution time on the x-th fastest client (where x is a fraction s of the number of affected clients) as a reference value for the abortion criterion, we ensure that no new load sequence has begun on this client with high probability by setting $s = \frac{1}{8}$ (instead of $s = \frac{1}{16}$). Provided that no new load sequence begins during the superstep, the factor between the fastest and the slowest normalized BSP process execution time on the particular clients is at most q^2. For a q-boundedness of $q = 1.6$ this yields $q^2 = 2.56$. We have actually chosen $r = 2$ because of our defensive choice of s.

Using these parameters, we could save 14% of the execution time averaged and even up to 25% in particular cases in comparison to the experiments with the PwoR algorithm; the savings in comparison to the experiments with no load balancing algorithm were even 32% averaged and up to 45% in isolated cases.

Results using SwJ. For the choice of the parameters, the same aspects as in the SwoJ case apply. In comparison to our experiments with the SwoJ algorithm we could save another 10% of the execution time averaged and even up to 27% in isolated cases; the savings in comparison to the experiments with no load balancing algorithm were even 39% averaged and up to 50% in particular cases.

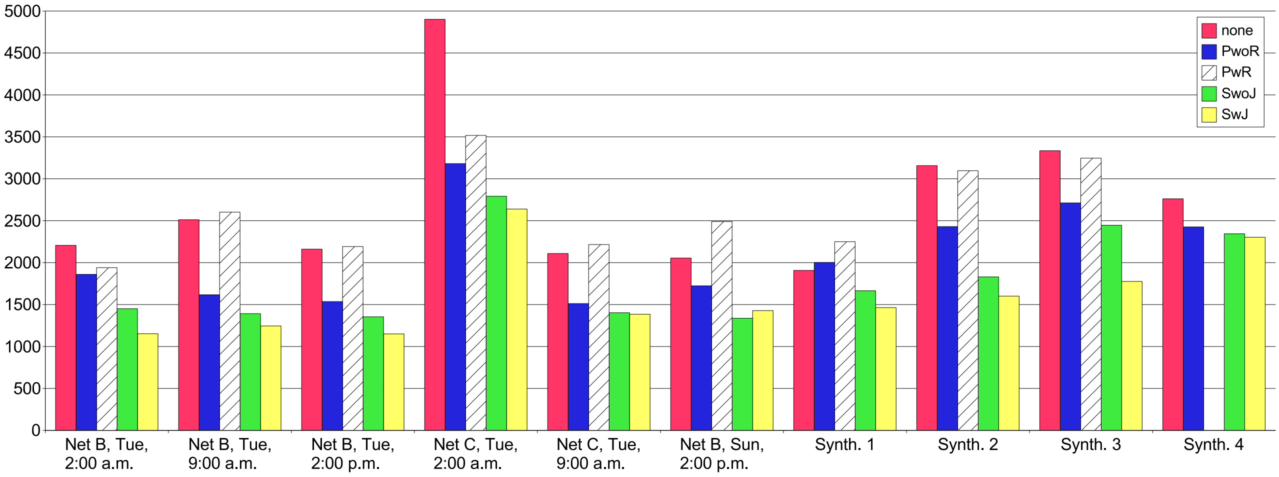

Fig. 7.2. *Running times depending on the load balancing algorithm.*

8. Conclusion. We have developed a web computing library that allows to execute BSP programs in a peer-to-peer network, utilizing only the computation power left over on the participating PCs. It features migration and restoration of the BSP processes in order to rebalance the load and increase fault tolerance because the available computation power fluctuates and computing nodes may join or leave the peer-to-peer system at any time.

We have implemented and analyzed different load balancing strategies for PUBWCL. The load balancing strategy SwJ performs better than SwoJ which, in turn, performs better than PwoR (cf. Fig. 7.2). In comparison to using no load balancing, we can save up to 50% of the execution duration using SwJ.

Due to security and portability reasons one has to use a virtual machine like Java's one, so a performance drawback cannot be avoided. The slowdown depends on the type of the BSP algorithm.

In order to further improve PUBWCL, work is in progress to realize PUBWCL as a pure peer-to-peer system in order to dispose of the bottleneck at the server, and to replace Java RMI in PUBWCL with a more efficient, customized protocol.

Additionally, we are working on an extension of PUBWCL which allows redundant execution of BSP processes, i. e., processes are started redundantly, but only the results of the fastest one are committed whereas the remaining processes are aborted when the first one completes. This will allow us to improve the load balancing strategies by starting additional instances of slow BSP processes on faster clients instead of just restarting them. Since, using the SwJ algorithm, typically only a very small fraction of the BSP processes was restarted, this would mean only a low overhead but would significantly reduce the probability that they would have to be restarted another time.

Acknowledgement. Partially supported by DFG-SFB 376 “Massively Parallel Computation” and EU IST-2004-15964 (AEOLUS).

REFERENCES

[1] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, 2004.

[2] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping, *The Paderborn University BSP (PUB) library*, Parallel Computing, 29 (2003), pp. 187–207.

[3] *distributed.net.* `http://www.distributed.net/`

[4] S. Fünfrocken, *Transparent migration of Java-based mobile agents*, in Mobile Agents, 1998, pp. 26–37.

[5] J. Gehweiler, *Entwurf und Implementierung einer Laufzeitumgebung für parallele Algorithmen in Java*, Studienarbeit, Universität Paderborn, 2003.

[6] J. Gehweiler, *Implementierung und Analyse von Lastbalancierungsverfahren in einer Web-Computing-Umgebung*, Diplomarbeit, Universität Paderborn, 2005.

[7] *Great internet mersenne prime search (GIMPS).* `http://www.mersenne.org/`

[8] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Computing, 24 (1998), pp. 1947–1980.

[9] J. Hromkovic and W. M. Oliva, *Algorithmics for Hard Problems*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[10] *JavaGgo RMI.* `http://www.joachim-gehweiler.de/en/software/javago.php`

[11] S. Leonardi, A. Marchetti-Spaccamela, and F. Meyer auf der Heide, *Scheduling against an adversarial network*, in SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, ACM Press, 2004, pp. 151–159.

[12] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau, *Delta execution: A preemptive Java thread migration mechanism*, Cluster Computing, 3 (2000), pp. 83–94.

[13] *The Paderborn University BSP library.* `http://wwwcs.uni-paderborn.de/~pub/`

[14] *The Paderborn University BSP-based Web Computing Library.* `http://wwwcs.uni-paderborn.de/~pubwcl/`

[15] T. Sakamoto, T. Sekiguchi, and A. Yonezawa, *Bytecode transformation for portable thread migration in Java*, in ASA/MA, 2000, pp. 16–28.

[16] L. F. G. Sarmenta, *An adaptive, fault-tolerant implementation of BSP for Java-based volunteer computing systems*, in Lecture Notes in Computer Science, vol. 1586, 1999, pp. 763–780.

[17] T. Sekiguchi, H. Masuhara, and A. Yonezawa, *A simple extension of Java language for controllable transparent migration and its portable implementation*, in Coordination Models and Languages, 1999, pp. 211–226.

[18] *Search for extraterrestrial intelligence (SETI@home).* `http://setiathome.berkeley.edu/`

[19] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, *Portable support for transparent thread migration in Java*, in ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, London, UK, 2000, Springer-Verlag, pp. 29–43.

[20] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

Edited by: Przemysław Stpiczyński.
Received: March 31, 2006.
Accepted: May 28, 2006.