Efficient Computations on fault-prone BSP machines*

Spyros C. Kontogiannis[†,§] Grammati E. Pantziou[§] Paul G. Spirakis[†,§]

Abstract

In this paper general simulations of algorithms designed for fully operational BSP machines on BSP machines with faulty or unavailable processors, are developed. The fail-stop model is considered for the fault occurrences, that is, if a processor fails or becomes unavailable, it remains so until the end of the computation. The faults are random, in the sense that a processor may fail independently with probability at most a, where a is a constant.

Two possible settings for fault occurrences are considered: the static case where the faults are static (the faulty or unavailable processors are already known at the beginning of the computation), and the dynamic case where the processors may become faulty or unavailable during the computation. In the case of static faults, a simulation of an n-processor fault-free BSP machine on a faulty n-processor BSP machine is presented with constant slowdown per local computation step and $\mathcal{O}(\log n \cdot \max\{L, g\})$ slowdown per communication step, given that a preprocessing has been done that needs $\mathcal{O}(n/\log n \cdot \max\{L, g\})$ time (L and g are the parameters of the simulating BSP machine).

In the case of dynamic faults, a simulation of an n-processor fault-free BSP machine on an $cn \cdot \log n$-processor faulty BSP machine is presented. No dynamic faults may occur during certain periods of the simulation. The simulations are randomized and Monte Carlo: they are guaranteed to be correct with high probability, and the time bounds always hold. To our knowledge, no previous work on the fault tolerance of the BSP model exists.

*This work was partially supported by the ESPRIT LTR ALCOM-IT (contract No. 20244) and GEPPCOM (contract No. 9072).

(†) Computer Engineering and Informatics Department, Patras University, 26500 Rion, Patras, Greece.

(§) Computer Technology Institute, Kolokotroni 3, 26221 Patras, Greece. E-mail:{kontog,pantziou,spirakis}@cti.gr

1 Introduction

The *Parallel Random Access Machine (PRAM)* is one of the earliest attempts to model a parallel machine. It combines the power of parallelism with the simplicity of single RAM processing elements, but ignores some realistic features of parallel machines, such as the synchronization and the communication mechanisms. *PRAM* has proven to be a very convenient cost model of parallel computation, mostly because it abstracts away some machine oriented features (e.g. the communication requirements and synchronization issues) of parallel computing. The model of parallel computation known as the *Bulk-Synchronous Parallel (BSP)* machine, which has attracted much attention in recent years, deals explicitly with the notion of communication and synchronization among computational threads. BSP has been proposed by L. Valiant ([V90a, V90b]) as a bridging model between software and hardware, and as a unified framework for the design, analysis, and programming of general purpose parallel computing systems, which permits the performance of those systems to be analysed and predicted in a precise way. The BSP model tries to capture the communication cost while keeping the parallel machine features hidden from the designer, and seems to be the new prevailing model in the parallel algorithms community.

BSP, like almost all the existing cost models of parallel computation, refers to ideal, fully reliable machines. In addition, efficient BSP computations imply a minimization of redundancy in the computation and therefore, very little space for fault tolerance is left. This means that certain BSP computations may become very inefficient, or even faulty in the presence of some processor or communication faults. In this paper we address the issue of fault tolerance in BSP computations. We develop general simulations supporting fault-free BSP computations in environments

where not all processors are operational or available to the computation at each step. Because of the nature of the BSP model, we consider possible faults or unavailability of processing elements, while the communication network is considered to be absolutely reliable.

For the fault occurrences, the *fail-stop model* is considered, that is, if a processor fails or becomes unavailable, it is considered to remain in this status, until the end of the current computation. We consider that faults are random, that is, a processor may fail independently and with probability less than or equal to a, where a is a constant to be specified in the sequel. Moreover, we consider simulations for two different cases: the static case, where the faulty or unavailable processors are already known at the beginning of the computation and no processor changes its status afterwards, and the dynamic case, where each processor may fail or become unavailable with a fixed probability during the computation (except for some critical periods during the computation), and remains so until the end of the computation.

1.1 Our results

Our main results concerning simulations of general BSP computations under static or dynamic processor faults, are the following:

- An n-processor BSP machine with a bounded fraction of faulty processors can simulate its fault-free counterpart in a step-by-step fashion with constant slowdown per local computation step and $\mathcal{O}(\log n \cdot \max\{L, g\})$ slowdown per communication step, given that a preprocessing has been done that needs $\mathcal{O}((n/\log n) \cdot \max\{L, g\})$ time w.h.p. L and g are the parameters of the simulating BSP machine.

- A $cn \log n$-processor BSP machine, c is a constant, with a bounded fraction (less than 1/2) of its processors to sustain dynamic faults (real machine) correctly simulates w.h.p. an n-processor fault-free BSP machine (virtual machine) with a multiplicative slowdown of g plus an additive slowdown of $\mathcal{O}(g(h + \log n))$ per local computation superstep[1], and an $\mathcal{O}(\log\log n)$ multiplicative slowdown plus an $\mathcal{O}(g \log n + h \log h)$ additive slowdown per communication superstep. No dynamic faults may occur during certain periods of the simulation, and the simulation bounds hold for L appropriately small.

[1] h is the number of messages to be sent or received by any processor of the BSP machine during the next communication superstep, thus implying the implementation of an h-relation.

The necessity for fault-free periods of the simulation process exists because we consider general (including probabilistic or non-deterministic) BSP computations (cf. section 4). Note that although the existence of fault-free periods during the simulation prevents the application of the result in a setting where hardware (processor) faults may dynamically occur, it can be very well applied in a parallel operating system setting, where processors become dynamically unavailable to certain processes. In such a case, certain primitives may lock the currently operational processors, so that they remain available to the simulation process during certain critical periods, and unlock them, so that they can be in the availability of the operating system which can dynamically allocate each one of them with fixed probability, to other processes.

1.2 Related Work

To our knowledge, no previous work on the fault tolerance of the BSP model exists. Yet, there has been a lot of research done on the fault tolerance of other parallel computation models, and especially of $PRAM$ and the *Optical Communication Parallel Computer (OCPC)* model, which is equivalent to the 1-collision DMM. Kanellakis and Shvartsman ([KS92, KS91]) initiated the study of computations on a faulty $PRAM$ in the case of deterministic faults, considering the non-restartable and the restartable fail-stop models. Kedem, Palem and Spirakis ([KPS90]) and Kedem et al. ([KPRS91]) studied the case of probabilistic, dynamic faults considering again the fail-stop model. Chlebus, Gambin and Indyk [CGI94] designed efficient randomized *CRCW PRAM* simulations for both the dynamic and static case of memory faults, considering an arbitrary, constant fraction of memory cells to be faulty. Chlebus, Gasieniecc and Pelc ([CGP95]) presented a deterministic simulation of an n-processor *EREW PRAM* on an n-processor *EREW PRAM* prone to processor and memory faults. Recently, Berenbrink, Meyer auf der Heide and Stemann ([BMS96]) and Chlebus, Gambin and Indyk ([CGI96]) presented simulations of the *EREW PRAM* on DMM for various models of faults.

All these results on the fault-tolerance of the $PRAM$ cost model are not directly comparable to ours, because it is well known that BSP is a more realistic cost model of parallel computation, that also captures the communication and synchronization costs that are considered to be negligible in the case of $PRAM$. Additionally, most of the strategies applied in $PRAM$ exploit the feature of the Shared Memory to achieve the fault tolerance, which is not possible in the case of BSP.

The rest of the paper is organised as follows: In

section 2 we give the definition of the BSP model and some basic facts. In section 3 we present the simulation of a fault-free n-processor BSP machine by an n-processor BSP machine considering a bounded fraction of faulty processors. In section 4 we show how a $cn \cdot \log n$-processor BSP machine (c is a constant) with a bounded fraction of its processors to sustain dynamic faults, can simulate an n-processor fault-free BSP machine.

2 Preliminaries

The BSP Model

The BSP model consists of the following components: A collection of n processor/memory elements which are identified by unique numbers $1, 2, \ldots, n$, a communication infrastructure that takes over the point-to-point communication process, and a mechanism for efficient barrier synchronization of the processing elements.

A BSP computation proceeds in a succession of supersteps separated by synchronizations. For clarity we consider that each superstep consists of two major phases: the *Local Computation phase* and the *Communication phase*. During the local computation phase each processor performs a sequence of operations on data held locally at the beginning of the superstep, while the communication phase takes over the transmission of all the messages held in the outgoing windows of the processors, to their destinations via the communication infrastructure. At the end of each superstep, a barrier synchronization mechanism is activated by the communication infrastructure, to confirm the end of the current superstep. A BSP machine is characterised by the following two parameters:

- g - the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered. g is related to the time required to realise h-relations in continuous message usage in gh time steps [Mc94].

- L - the minimum time between two consecutive synchronization operations. Thus L is the minimum time for a superstep.

The running time of an algorithm on the BSP cost model depends on the number of supersteps it has to perform, in order to complete its task. Each superstep is charged

$$T_{\text{superstep}} = \max\{L, T_c + T_r\}$$

basic time steps, where T_c is the maximum number of basic computational operations executed by any processor during the superstep, and $T_r = gh$, where h is the maximum number of 1-word messages transmitted or received by any processor during the superstep. Note that for some $h_0 \geq 0$ that depends on the characteristics of the underlying communication infrastructure, any h-relation for which $h < h_0$, will be charged as an h_0-relation.

In order to simplify the description of the performance of BSP algorithms, we may assume that each superstep is either a pure local computation or a pure communication step ([GV94]). Therefore we have either local computation supersteps or communication supersteps and the cost of a superstep is either $\max\{L, T_c\}$ or $\max\{L, gh\}$. The following fact will be used in the time estimations of our techniques:

Fact 2.1 *There exists a BSP algorithm that broadcasts a k-word message to n processors that requires time at most $\mathcal{O}(\log n \cdot \max\{L, gk/\log n\})$. In particular, if $L \leq gk/\log n$, then the algorithm needs $\mathcal{O}(gk)$ time steps.*

For a proof of the fact the reader is referred to [GS96, BDM95].

In the sequel, we say that a property $\mathcal{P}$ depending on a natural number n holds *with high probability (w.h.p.)* if there exists a constant $c > 0$ such that $\mathcal{P}$ holds with probability at least $1 - n^{-c}$, for sufficiently large n. Finally, we shall denote by $[n]$ the set of integrals $\{1, \cdots, n\}$.

3 BSP computations under static faults

In this section, we show how to overcome processor faults on an initial n-processor BSP machine, which we shall call the *Virtual Machine ($\mathcal{VM}$)*, by specifying a fault-free part of it that is determined during a preprocessing phase, and simulating afterwards the fault-free operation of the initial machine (i.e., the operation of an ideal n-processor BSP machine). We shall call this realistic, healthy part of $\mathcal{VM}$, the *Real Machine ($\mathcal{RM}$)*. We consider that the faults are static, in the sense that they have already occurred before the start of the computation, and random, that is, a processor P_i of $\mathcal{VM}$ may fail independently with probability $p_i < a$, where a is a constant. It is also considered that if an operational processor attempts to communicate with a faulty processor then the communication network discards the request and acknowledges the requester. Finally, the non-restartable fail-stop model is considered for the processor failures ([KS92]).

For the sake of simplicity we consider that each processor in $\mathcal{VM}$ has a unique *Virtual Identification Number (Vid)* in the range $[1..n]$. Each processor in $\mathcal{VM}$ is called *virtual processor* and it is indicated by $P_i, i \in [n]$. A faulty virtual processor is called *inactive*, while a virtual processor that is operating properly is considered to be *active*. Our purpose is to determine in the preprocessing phase, the number m of active processors in $\mathcal{VM}$ that comprise $\mathcal{RM}$, and assign to each of them a unique *Real Identification Number ($\mathcal{R}id$)* in the range $[1..m]$. Each processor of $\mathcal{RM}$ is called *real processor* and it is indicated by Q_j, $j \in [m]$. Subsequently, $\mathcal{RM}$ takes over the simulation of the fault-free operation of the initial fault-prone machine, $\mathcal{VM}$. We first discuss the construction of the fault-free machine $\mathcal{RM}$ (subsection 3.1) and then the simulation of the initial machine, $\mathcal{VM}$, by the real machine, $\mathcal{RM}$ (subsection 3.2).

An issue that needs to be addressed in this discussion, is the way that the input is provided to the machine. Is the input provided only to the active processors after the preprocessing, or to all processors? As it has been discussed in related works for other models of parallel computation ([CGP95, CGI96]), it is hardly acceptable to assume that the input is provided only to the active processors after the preprocessing. Therefore, we assume that different parts of the input are stored in an encoded form in different processors of the fault-prone machine and, during the preprocessing the processors of $\mathcal{RM}$ will have to restore the input for the simulated algorithm. Methods of encoding and then retrieving the original information when only part of it is available, are known as *error-correcting codes*. A simple and popular method is the *Information Dispersal Algorithm - IDA* of Rabin ([R89]), and an $\mathcal{O}(\log^2 n)$-time *PRAM* version of this algorithm is discussed in [CGP95]. We may apply standard *PRAM* simulation on a *BSP* machine to obtain a *BSP* version of *IDA*. An alternative approach, is to implement Rabin's algorithm directly on the *BSP* model. Another method that one could apply, is the *Error-Correcting Expander Codes* of Sipser and Spielman ([SS94]). In this version of our work, we concentrate on the simulation itself, and we leave the issue of the input for the full paper.

3.1 Construction of the fault-free machine

We present the algorithm Construct-$\mathcal{RM}$ that determines a fault-free subset of the virtual fault-prone n-processor *BSP* machine. The algorithm consists of two phases. In the first phase, the number m of fault-free processors in $\mathcal{VM}$ is determined, while in the second phase, each one of these processors is assigned a unique $\mathcal{R}id$ in the range $[1..m]$.

Phase 1 /* bottom-up movement */
begin
  $\forall\, j$, P_j marks itself as a master processor.
  for $i = 1$ **to** $\log n$ each active processor P_j does:
  begin
    if P_j is a master processor
    then begin
      if $i \leq \text{loglog}\, n + 1$
      then P_j seeks extensively for its opponent
      in the proper target group
      else P_j makes k independent trials to locate it,
      or a slave processor that will lead to it.
      if P_j has located its opponent,
      then it challenges it for duel, and
        if P_j wins
        then P_j continues with the next round,
        else P_j marks itself as a slave and
        is out of the game.
      else P_j continues with the next round
    endThen
    else begin /* P_j is a slave processor */
      P_j waits until a processor challenges it, or
      the current round ends.
      if P_j is challenged by a processor,
      then it forwards the challenge to its parent,
      and acknowledges the challenging processor.
    endElse
  endFor /* End of the $\log n$ rounds of duels */
endPhase 1.

Figure 1: The bottom-up Phase of Construct-$\mathcal{RM}$.

In *Phase 1* (see figure 1), Construct-$\mathcal{RM}$ proceeds in a sequence of consecutive rounds of duels between virtual processors, until a unique processor survives. For a duel to take place, an active processor has to challenge another active processor. The winner of a duel is called a *master processor*, while the defeated processor is called a *slave processor*. The rule for winning in a duel is not significant, since the key point during this process is the election of any active processor to be the only global winner that will have survived from all these duels. Thus one may consider any rule for the duels. In this construction we select the active processor with the bigger $\mathcal{V}id$ to be the winner of the duel.

Each slave processor is hooked by the master processor that defeated it, thus constructing a tree structure. This is rather a dynamic tree construction, since, as it will be exhibited later on, in each round a master processor will have to seek for its counterpart and

fight it. The winning processor of a duel also keeps the score of the duel in a first-in-first-out structure, called *SCORES*.

In general, during the i^{th} round of duels, $i = 1, \cdots, \log n$, each master processor seeks for its opponent master processor, if any, in a target group of virtual processors of size 2^{i-1}. Note that in each target group, there exists at most one master processor. An internal node of the tree structure that is constructed, holds a replica of the winner of the duel between the virtual processors corresponding to its children, along with a time stamp indicating the round in which the duel took place. This might be seen as if some of the active processors (the winners of the duels in each round) climb up the tree structure.

When a duel occurs in a specific level (other than the level of the leaves), the challenging processor is not aware of the identity of its opponent (which is the root of the corresponding subtree in the binary tree structure), and it will thus have to seek for its counterpart in the corresponding target group of virtual processors, that consists of the leaves of the subtree in challenge. That's why this is rather a dynamic and not just a trivial binary tree structure.

In the first $\text{loglog}\, n + 1$ rounds the target groups have size $\leq \log n$ and a challenging processor may seek for its opponent in the corresponding target group extensively, in logarithmic time. For the rest of the rounds, the target group grows significantly, and an extensive search would be very expensive. Thus, a randomized technique is applied, where each master processor makes k consecutive trials at random to locate either its opponent (master processor), or a slave processor in the target group under question, that will lead to it.

Lemma 3.1 *The probability of the binary tree structure failing to be constructed successfully is bounded by:*

$$P_{\text{fail}} < \frac{1}{2}\left(\frac{n}{\log n} - 1\right) \cdot n^{-\frac{(1-a)c-1}{\ln 2}} = \mathcal{O}(n^{-c_1})$$

where c_1 is an appropriately chosen constant that depends on a.

Proof: The proof of this technical lemma is based on the application of Chernoff Bounds ([MR95, AS92]) on the failure probability of the tree structure, and is omitted here due to lack of space. The reader is referred to the full version of this paper ([KPS97]) for a complete presentation of the proof. ■

At the end of *Phase 1*, the global winner will already know the number m of active processors in $\mathcal{VM}$, which will also be its own $\mathcal{R}id$ since this is the active processor of $\mathcal{VM}$ with the biggest $\mathcal{V}id$. During *Phase 2* (see figure 2), the global winner first assigns to its $\mathcal{R}id$ the value m, and then initiates a top-down flow of information in the binary tree structure having already been constructed, during which all active processors are assigned their $\mathcal{R}id$'s.

```
Phase 2 /* top-down movement */
begin
  for i:=1 to log n each active processor does:
  begin
    if P_j is a master processor and the SCORES
    structure is not empty,
    then it pops the next duel from SCORES and
    sends a reactivation message to the defeated
    processor, along with its new Rid
    else   if P_j receives a reactivation message,
           then it gets a new Rid and becomes
           master.
  endFor
endPhase 2.
```

Figure 2: The top-down Phase of Construct-$\mathcal{RM}$.

Lemma 3.2 *The algorithm Construct-$\mathcal{RM}$ computes the number of active processors in $\mathcal{VM}$ and assigns to them their $\mathcal{R}id$'s in $\mathcal{O}(\log^2 n \cdot \max\{L, g\})$ time, w.h.p.*

Proof: We first address the number of steps that the algorithm needs, while the failure probability is assured to be sufficiently small by the previous technical lemma.

In the first phase, each master processor looks for its counterpart for duel in a specific target group, the size of which is related to the level of the processor in the binary tree structure. If a master processor is located at a level $i \leq \text{loglog}\, n + 1$, then the search will be extensive. In the worst case of these initial $\text{loglog}\, n + 1$ rounds, there will be at most 2^i messages that will be sent by a master processor to processors of its target group, while each master or slave processor may receive at most $\mathcal{O}(1)$ messages in round i. Considering each of these rounds as a single superstep, for round i the time cost will have $T_c =$ negligible, and $T_r = g \cdot 2^i$, while the amount of time in this preliminary phase of *Phase 1* will be:

$$T_{\text{pre}} = \sum_{i=0}^{\text{loglog}\, n} (\max\{L, g2^i\})$$

In the case that a master processor is located at a level $i > \text{loglog}\, n + 1$ of the binary tree structure, then this processor uses a randomized technique to locate its opponent, since an extensive search operation

would introduce a prohibitive cost. More specifically, it makes k independent trials, until it specifies its opponent in the corresponding target group, or a slave processor that will lead it to the master processor of the target group. If a master processor neither manages to locate its opponent in the target group, nor is challenged by another (master) processor of the target group, it concludes that no active processor exists in the target group under question and proceeds with the next round.

The cost of communication during the construction of a specific level i of the binary tree, obviously depends on the number of messages that any active processor has to either send or receive. Since a master processor makes at most k trials to locate its opponent in the corresponding target group and each active processor may receive up to k messages, a k-relation has to be implemented, that requires time $T_r = g \cdot \max\{k, h_0\} + L$, and since the local computation cost is negligible, the total time required for a master processor to locate its opponent or a slave processor that will lead it to its opponent (master processor) for, e.g. $k = c \cdot \log n$, is $\max\{L, cg \cdot \log n\}$.

In the case that a master processor hits a slave processor in the target group, then the challenge will be forwarded to the master processor of this group by exploiting the already constructed part of the binary tree for internal communication in the target group. This internal communication will cost at most $\log 2^i \leq \log n$ supersteps per round. Since the h-relations that need to be implemented here are such that $h = 1$, each superstep needs $\mathcal{O}(\max\{L, g\})$ time.

Thus, the total time of the randomized part of *Phase 1* will be

$$\begin{aligned} T_{\text{rand}} &= \sum_{i=\log\log n+1}^{\lceil \log n \rceil} (\max\{L, cg \cdot \log n\} + \\ &\qquad + \log n \cdot \max\{L, g\}) \\ &= \mathcal{O}(\log^2 n \cdot \max\{L, g\}) \end{aligned}$$

As for the second phase of the algorithm, this is implemented in $\mathcal{O}(\log n)$ steps, since the information has to reach the leaves of the binary tree, and there is no serious delay of the wave front of information in each internal node. Moreover, in each step, the h-relations that need to be implemented are such that h is a small constant. ■

3.2 The Simulation Process

After having determined the parameters of $\mathcal{RM}$, each real processor only knows the amount of real processors (that is, the amount of active processors in $\mathcal{VM}$) and its own $\mathcal{R}id$. Now the task is to determine a procedure that will simulate the fault-free operation of the fault-prone n-processor BSP machine ($\mathcal{VM}$) on the fully operational m-processor BSP machine ($\mathcal{RM}$). This task is divided in two major subtasks: The simulation of the local computation and the communication of the virtual processors.

Local Computation Phase simulation

The simulation of the local computation phase of a virtual processor is rather simple, in the sense that each active processor takes over the operation of $\lceil \frac{n}{m} \rceil$ virtual processors. Specifically, the task of a virtual processor P_j is simulated by the real processor Q_l, where j and l are correlated by an easily computable function. For example, in the following we will consider that $l = j \bmod m$. Thus a real processor Q_l knows in advance the virtual processors whose the operation it is expected to simulate. Finally, Q_l may compute in constant time the $\mathcal{R}id$ of the real processor that is responsible for the simulation of a virtual processor which is a target processor for it.

Communication Phase simulation

In the communication phase things are much more complicated, since we have to specify the actual target of a message that needs to be sent by a virtual processor P_i to a virtual processor P_j. Recall that the communication infrastructure recognizes the processing elements by their unique addresses, which may be assumed to be the $\mathcal{V}id$'s of the processors in $\mathcal{VM}$. But what is the physical address of a real processor that is responsible for the operation of a virtual processor which is the target of a specific message?

The solution to this problem is the exploitation of the binary tree structure, built during the execution of Construct-$\mathcal{RM}$, for the determination of the actual identity (which is the $\mathcal{V}id$) of a specific real processor. This solution goes up the tree until it finds a processor whose subtree is large enough to contain the real processor in question. The method uses message passing for the determination of the identity of the real processors in question, but there is an overhead which may be as large as $\mathcal{O}(n)$, in extreme situations (e.g. when all the active processors of the right subtree of the root want to access some real processors in the left subtree). Obviously, in such cases there is a bottleneck at the root and this solution is as bad as a trivial all-to-all broadcast operation that implies the implementation of an n-relation.

In order to deal with such tricky situations, we augment *Phase 2* of Construct-$\mathcal{RM}$ and transfer some routing information to some special processors of the tree structure. More specifically we assure that there will be no need for any request to go up the tree, further than level $f(n)$, $0 \leq f(n) \leq \log n$ where the

value of $f(n)$ directly affects the trade-off between the preprocessing and routing time, and will be specified in the analysis of this augmented version of *Phase 2*.

According to this routing scheme, each processor sends its request up in the tree, until it reaches level $f(n)$, and the corresponding processor either contains the requested processor in its own subtree, or is capable of indicating another processor at the same level, whose subtree will contain it.

More specifically, each active processor keeps a new structure, *ACKNOWLEDGED*, of physical addresses for the real processors being at the same level of the tree. This structure is updated during the top-down movement, starting from the root, down to level $f(n)$. As shown in figure 2, each master processor recursively reactivates all the processors held in its *SCORES* structure during the top-down movement. In the augmented version of *Phase 2* each of its children, apart from the reactivation message, will also get a replica of its parent's *ACKNOWLEDGED* structure. This is an 1-relation of at most $(\frac{n}{2^{i+1}}+1)$-word messages when going from level $i+1$ to level i, because each newly reactivated processor communicates with its own parent, and there are at most $\frac{n}{2^{i+1}}$ active processors in level $i+1$ that already have been included to the parent's *ACKNOWLEDGED* structure.

Furthermore, each parent that reactivates an active processor informs all the processors in its *ACKNOWLEDGED* structure for this reactivation, thus implying an $(\frac{n}{2^{i+1}})$-relation among the processors that already know each other.

Finally, each newly reactivated processor is updated by its parent for the rest of the newly reactivated processors in the tree (at most $\frac{n}{2^{i+1}}$ because of the special features of the tree structure), implying once more an 1-relation of at most $(\frac{n}{2^{i+1}}+1)$-word messages.

When the top-down wave front of reactivation reaches level $f(n)$, it keeps going down without affecting the *ACKNOWLEDGED* structures of the newly reactivated processors anymore.

As for the time for routing during a communication phase, this strongly depends on $f(n)$, the level where the update of the *ACKNOWLEDGED* structures has stopped. Suppose that all the processors in level $f(n)$ already know each other's physical address and we want to perform the routing before the next Communication Phase. Each processor in level $f(n)$ is aware of the physical addresses of $(\frac{n}{2^{f(n)}}-1)$ processors out of its own subtree, and there are $2^{f(n)}$ processors at its leaves. Thus at most $\min\{2^{f(n)}, n-\frac{n}{2^{f(n)}}-2^{f(n)}\}$ requests for addresses will need to be cross-referenced to some other processor of level $f(n)$. Moreover, each processor in level $f(n)$ may receive at most $h\cdot 2^{f(n)}$ requests for some of its leaves, considering that $\mathcal{VM}$ has to implement an h-relation.

From the above discussion, and after taking into account that even an 1-relation of $(\frac{n}{2^{i+1}})$-word messages costs the same as an $(\frac{n}{2^{i+1}})$-relation, according to Fact 2.1, the following lemma holds:

Lemma 3.3 *Assuming that the routing information has gone down to level $f(n)$, the preprocessing and routing times of our simulations for the static case are given by:*

$$\begin{aligned} T_{\text{augm\_phase 2}} &= \mathcal{O}\Big(\frac{n}{2^{f(n)}}+f(n)\Big)\cdot\max\{L,g\}, \\ T_{\text{route}} &= \mathcal{O}\Big(h\cdot 2^{f(n)}+f(n)\Big)\cdot\max\{L,g\} = \\ &= \mathcal{O}\Big(h\cdot 2^{f(n)}\Big)\cdot\max\{L,g\}. \end{aligned}$$

The extensive proof of Lemma 3.3 will be shown in the full version of the paper. According to this lemma, an $\mathcal{O}(2^{f(n)})$ multiplicative slowdown per communication phase is induced, while the preprocessing time is the sum of the times $T_{\text{phase 1}}$ and $T_{\text{augm\_phase 2}}$. For example, if we consider that $f(n)=\text{loglog}\,n$ then we have: $T_{\text{route}}=\mathcal{O}(\log n)$, while $T_{\text{preprocessing}}=\mathcal{O}(\frac{n}{\log n}+\text{loglog}\,n+\log^2 n)$, and if $f(n)=\log(\frac{n}{\log^c n})$, $T_{\text{route}}=\mathcal{O}(\frac{n}{\log^c n})$ and $T_{\text{preprocessing}}=\mathcal{O}(\log^c n+\log(\frac{n}{\log^c n})+\log^2 n)$.

Finally notice that for consecutive communication steps, the routing overhead diminishes because each processor of level $f(n)$ knows more and more processors located at subtrees other than its own one.

Theorem 1 *An n-processor BSP machine with a bounded fraction of faulty processors can simulate its fault-free counterpart in a step-by-step fashion with constant slowdown per local computation step and $\mathcal{O}(2^{f(n)}\cdot\max\{L,g\})$ slowdown per communication step, given that a preprocessing has been done that needs $\mathcal{O}((\frac{n}{2^{f(n)}}+f(n)+\log^2 n)\cdot\max\{L,g\})$ time w.h.p. L and g are the parameters of the simulating BSP machine, $1\le f(n)\le\log n$.*

Proof: The above discussion, along with Lemmas 3.1, 3.2 and 3.3, completes the proof of Theorem 1. ■

4 BSP computations under Dynamic Faults

In this section we present simulations of an n-processor *BSP* machine on an m-processor *BSP* machine, in the presence of dynamic faults. In this case, instead of determining the healthy part of the initial machine and then trying to simulate its fault-free operation, we assume that we have an m-processor faulty *BSP*

machine ($\mathcal{RM}$) that will have to simulate the fault-free operation of an n-processor BSP machine ($\mathcal{VM}$), where $m = cn \cdot \log n$, and c is an appropriately chosen constant. The main idea is to assure the fault-free operation of $\mathcal{VM}$, at the expense of some extra processing elements.

4.1 Model of Simulation

The $cn \log n$ processors of $\mathcal{RM}$ are organised into n groups G_i, $i = 1, \cdots, n$, each of $c \log n$ processors. The task of group G_i is to simulate the fault-free operation of virtual processor P_i of $\mathcal{VM}$. Real processor Q_j (with j in the range $\{0, \cdots, m-1\}$) will belong to group G_i, where $i = j \bmod n$, while its cardinal number in G_i will be $l = j \div n$[2]. Alternatively, Q_j may also be indicated as $Q_{i,l}$.

We focus on the simulation of a virtual superstep of $\mathcal{VM}$ on $\mathcal{RM}$. For clarity of the description, we assume that we have local computation supersteps and communication supersteps. As for the model of faults, we consider again the fail-stop model, and we assume that the faults are dynamic, in the sense that each processor may fail or become unavailable during the simulation independently and with probability less than or equal to a, where a is a constant less than $1/2$. We also assume that there are certain periods during the simulation, where no faults may occur, or the processors must remain available to the simulation process. We shall call these periods *critical periods*. As we argue in the sequel, in the case of probabilistic and non-deterministic computations, the need for the correct dissemination of the appropriate information among the processors in each group so that no inconsistencies occur in the simulation of $\mathcal{VM}$, necessitate the existence of these critical periods. Note that although the existence of the critical periods complicates the application of the simulation in a setting where processor faults may occur dynamically, it can be very well applied in a parallel operating system setting where processors may become unavailable to certain processes. In such a case, certain primitives may lock the currently operational processors, so that they remain available to the simulation process for the duration of these safe periods, and unlock them, so that they can be in the availability of the operating system which can dynamically allocate each of them, with fixed probability, to other processes.

When the simulation starts, each active processor in G_i, knows the initial state of processor P_i, $i = 1, \cdots, n$. In the course of the simulation, if an active processor in G_i has all the information necessary for the next virtual superstep, i.e., is aware of the current state of P_i, then it will be called *informed*.

[2]The operations $\div$ and mod are considered to be the ordinary integer operations.

Local computation superstep simulation

We consider two stages in the simulation of such supersteps. In the first stage, the local computation is taking place and arbitrary processor faults are allowed. The second stage is a critical period, during which the active processors interchange information, so that all active processors in the same group are in the same state by the end of the simulation of the local computation superstep. The description of the stages is as follows.

Stage 1: All active processors in G_i simulate the task of processor P_i, $i \in [n]$. Note that in the case of probabilistic or non-deterministic computations, after the end of the simulation of the local computation phase of $\mathcal{VM}$, not all active processors of the same group are necessarily in the same state. For the simulation to work, we should guarantee that all active processors in G_i end up in the same state. Therefore, we need the following second stage in the local computation superstep simulation, for tuning among the active processors of the same group.

Stage 2: We assume that a critical period of the simulation starts during which no faults may occur, and the following are taking place. All active processors of G_i participate in a sequence of duels and a tree structure of the active processors is created with the root being the global winner of all duels. This is done in a way similar to the one in the case of static faults, but the challenging processors seek for their opponents always extensively in the target group, since the target groups are small enough. The global winner uses the tree structure to broadcast to all currently active processors the contents of its memory (including the contents of the outgoing windows) that have been changed during the simulation of the latest local computation phase of $\mathcal{VM}$. Thus, at the end of this stage all active processors in G_i are informed.

Communication Phase Simulation

Again the simulation is divided into two stages. In the first stage, the communication of messages between the groups of $\mathcal{RM}$ is taking place, and the processors may die dynamically. In the second stage, which is a critical period of the simulation, the processors interchange the messages received at the previous stage, so that all active processors are informed by the end of the simulation of the communication superstep.

Stage 1: Suppose that at the communication superstep of $\mathcal{VM}$, the message s held in the outgoing window of the virtual processor P_i is transmitted to processor P_j. Since we keep as an invariant that each

time the simulation of the communication superstep starts, all active processors of $\mathcal{RM}$ are informed, the message s is held in the outgoing window of each currently active processor $Q_{i,k}$, $k \in [c \cdot \log n]$. Then, for all $k \in [c \cdot \log n]$, if processor $Q_{i,k}$ is informed and $Q_{j,k}$ is active, then the message s is transmitted to processor $Q_{j,k}$.

Note that at the end of this first stage of the communication superstep of $\mathcal{VM}$ there might be no processor in G_j that is completely informed. This is due to the fact that there might be no processor in G_j having received all the messages destined for processor P_j. Yet, the following lemma holds:

Lemma 4.1 *For each virtual processor P_j and for each message s received by P_j at the communication superstep of $\mathcal{VM}$, there is a processor in G_j that received s, w.h.p.*

For a proof of this technical lemma the reader is referred to the full version of this work ([KPS97]).

Since by Lemma 4.1 for each message destined for P_j there is at least one processor in G_j that received the message, all the messages are disseminated to all the processors in the group as follows:

Stage 2: Consider that a critical period of the simulation starts during which no dynamic faults may occur. A tree structure of the currently active processors is built similar to the structure built in the simulation of the local computation superstep. The tree structure is used for all the messages held by the active processors (leaves of the tree) to be sent to the global winner. Thus, after this step, the global winner holds all the messages sent to the group G_j at the current communication superstep. Then the global winner broadcasts these messages to all active processors.

Theorem 2 *A $cn \log n$-processor BSP machine, c is a constant, with a bounded fraction (less than $1/2$) of its processors to sustain dynamic faults (real machine) correctly simulates w.h.p. an n-processor fault-free BSP machine (virtual machine) with a multiplicative slowdown of g plus an additive slowdown of $\mathcal{O}(g(h + \log n))$ per local computation superstep*[3]*, and an $\mathcal{O}(\log\log n)$ multiplicative slowdown plus an $\mathcal{O}(g \log n + h \log h)$ additive slowdown per communication superstep. No dynamic faults can take place during certain periods of the simulation, and the simulation bounds hold for L appropriately small.*

Proof (sketch): The correctness of the simulation comes from its description and the proof of Lemma 4.1. To see the time requirements of the simulation we look at each one of the four stages separately. Let $\max\{L, T_c\}$ and $\max\{L, gh\}$ be the times that a Local Computation suprestep, and a Communication superstep of $\mathcal{VM}$ need respectively (i.e., at most h messages are sent or received by any processor).

Local computation superstep simulation. Stage 1 needs at most $\max\{L, T_c\}$ time for all the processors in group G_i to simulate processor P_i of $\mathcal{VM}$, $i \in [n]$.

As it can be seen from the proof of Lemma 3.1, and the fact that each group G_i has $\mathcal{O}(\log n)$ processors, the amount of time that the tree structure needs to be built is

$$\sum_{i=0}^{\log\log n} (\max\{L, g \cdot 2^i\})$$

Note that for appropriately small L, the tree can be built in $\mathcal{O}(g \cdot \log n)$ time steps. In the sequel of stage 2, the global winner broadcasts the contents of its memory that have been changed during stage 1 and the contents of its outgoing window to all active processors. This information can be included in an at most $(T_c + h)$-word message, and applying Fact 2.1, this stage requires $\log\log n \cdot \max\{L, \frac{g \cdot (T_c + h)}{\log\log n}\}$ time steps. Note that for appropriately small L, stage 2 needs $\mathcal{O}(g(T_c + \log n + h\})$ time steps.

Communication superstep simulation. From the description of stage 1, it is clear that each processor in any group G_i sends and/or receives at most h messages. Thus, stage 1 needs at most $\mathcal{O}(\max\{L, gh\})$ time steps. Stage 2 needs

$$\sum_{i=0}^{\log\log n} (\max\{L, g2^i\})$$

steps for the tree structure to be built up and the global winner to be elected. After that, each processor sorts the (at most) h messages it has received, according to the identification number of the processors that the messages are coming from, and then it sends them to the global winner as follows: At the first step, each leaf processor sends the messages that it has received as an (at most) h-word message to its parent in the tree (if different than itself). At step i, each internal processor of level i merges the sorted list of messages it has received during the previous step, together with its own sorted list of messages, eliminates duplicated messages, and sends the resulting at most h-word message to its parent processor. The procedure completes within at most $\mathcal{O}(\log\log n)$ communication supersteps. Since at most h-word messages are sent, each superstep requires $\mathcal{O}(\max\{L, gh\})$ time. Note that the computation supersteps involved require in total, at most $\max\{L, h\}$ time steps (time required to merge two sorted lists [CLR90]).

Finally, the last task of stage 2, i.e., the broadcasting of all h messages to all active processors, needs

[3] h is the number of messages to be sent or received by any processor of the BSP machine during the next communication superstep.

$\log\log n \cdot \max\{L, \frac{gh}{\log\log n}\}$ time steps (by Fact 2.1). Note that for appropriately small L, stage 2 needs $\mathcal{O}(gh \cdot \log\log n + g \cdot \log n + h \cdot \log h)$ time steps. ■

Since we do not allow any shared memory in our model and additionally, we want to support simulations of general (including probabilistic and non-deterministic) computations, we need to assume that there is a certain time period in our simulation of the local computation supersteps, in order to guarantee that the results of one of the active processors will be disseminated to the others, so that all processors of the same group are in the same state and therefore, they will transmit and receive the same messages during the subsequent communication superstep. Otherwise, we will not be able to take advantage of the large number of processing elements in the simulating machine (real machine) to reduce the per-step overhead in the simulation of the fault-free machine.

We may avoid the fault-free period during the simulation of the communication superstep, given that we increase the number of the messages that each processor transmits and/or receives. It is clear that if we allow for each processor in a group to send its messages to all processors of the target group (i.e., $ch \cdot \log n$-relations are implemented), then there is no need for the Stage 2 in the simulation of the communication superstep, and therefore, for the fault-free period during the simulation. But, in this way, we have a logarithmic multiplicative overhead in the simulation of the communication supersteps.

References

[AS92] N. Alon and J. Spencer. "The Probabilistic Method", Wiley Interscience, New York, 1992.

[BDM95] A. Baumker, W. Dittrich, and F. Meyer auf der Heide, "Trully efficient parallel algorithms: c-optimal multisearch for an extension of the *BSP* model", in the Proc. of the *3rd Ann. European Symp. on Algorithms*, 1995, Springer Verlag LNCS 979, pp. 17-30.

[BMS96] P. Berenbrink, F. Meyer auf der Heide and V. Stemann, "Fault-tolerant shared memory simulations", in the Proc. of the *13th Ann. Symp. on Theoretical Aspects of Computer Science (STACS'96)*, Springer Verlag, pp. 181-192.

[CGI96] B. Chlebus, A. Gambin, and P. Indyk, "Shared-Memory Simulations on a Faulty DMM", in the Proc. of *Intl. Colloquium on Automata, Languages and Programming*, 1996.

[CGI94] B. Chlebus, A. Gambin, and P. Indyk, "PRAM computations resilient to memory faults", in Proc. of the *2nd Ann. European Symp. on Algorithms (ESA'94)*, Springer Verlag LNCS 855, pp. 401-412.

[CGP95] B. Chlebus, L. Gasieniec, and A. Pelc, "Fast Deterministic Simulation of Computations on Faulty Parallel Machines", in Proc. of the *3rd Ann. European Symp. on Algorithms*, 1995, Springer Verlag LNCS 979, pp. 89-101.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest, "Introduction to Algorithms", *MIT Press*, 1990.

[GS96] A. Gerbessiotis and C. Siniolakis, "Communication efficient data structures on the *BSP* model with Applications", Oxford Univ. Technical Report, PRG-TR-13-96, May 1996.

[GT92] M. Gereb-Graus and T. Tsantilas, "Efficient optical communication in parallel computers", in the Proc. of the *4th Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA'92)*, 1992, pp. 41-48.

[GV94] A. Gerbessiotis and L. Valiant, "Direct Bulk-Synchronous Parallel Algorithms", *Journal of Parallel and Distributed Computing*, 22, pp. 251-267, 1994.

[KPRS91] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis, "Combining tentative and definite executions for very fast dependable parallel computing", in Proc. of the *23rd Ann. ACM Symp. on Theory of Computing*, 1991, pp. 381-390.

[KPS90] Z. Kedem, K. Palem, and P. Spirakis, "Efficient robust parallel computations", in Proc. of the *22nd Ann. ACM Symp. on Theory of Computing*, 1990, pp. 138-148.

[KPS97] S. Kontogiannis, G. Pantziou, P. Spirakis, "EfficientComputations on fault-prone *BSP* machines", CTI-TR 97.01.06. Also available through `http://www.ceid.upatras.gr/~kontog/bsp.ps`

[KS91] P. Kanellakis and A. Shvartsman, "Efficient parallel algorithms on restartable fail-stop processors", in the Proc. of the *10th Ann. ACM Symp. on Principles of Distributed Computing*, 1991, pp. 23-36.

[KS92] P. Kanellakis and A. Shvartsman, "Efficient parallel algorithms can be made robust", *Distributed Computing*, 5, 1992, pp. 201-217.

[Mc94] W.F. McColl, "Scaleable Parallel Computing: A grand unified theory and its practical development", In the Proc. of *IFIP World Congress*, 1, pp. 539-546, Hamburg, August, 1994.

[MR95] R. Motwani and P. Raghavan. "Randomized Algorithms", Cambridge University Press 1995.

[R89] M.O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance", *Journal of ACM*, 36 (1989), 335-348.

[SS94] M. Sipser and D. Spielman, "Expander codes", in the Proc. of *35th Ann. Symp. on Foundations of Computer Science*, (1994), pp. 566-576.

[V90a] L. Valiant. "A bridging model of parallel computation", *Communications of the ACM*, 33(8):103-111, August 1990.

[V90b] L. Valiant. "General purpose parallel architectures", In *Handbook of Theoretical Computer Science*, J. van Leeuwen ed., North Holland, 1990.

Appendix

Failure Probability evaluation

Let X_i be a random variable indicating the result of the i^{th} trial of processor P_j to locate its opponent while being at level r, in the target group of size 2^{r-1} (X_i is 1 during the i^{th} trial, if P_j locates its opponent or a slave that will lead to it, 0 otherwise). Since the trials are considered to be independent and the probability of P_j being inactive is $p_j < a$, we have:

$$Pr[X_i = 1] > 1 - a,\ Pr[X_i = 0] < a$$

Let Y be a random variable for P_j dependent on X_i's by the following relation:

$$Y = \sum_{i=1}^{k} X_i$$

Then it follows that $Pr[P_j\ fails\ to\ locate\ its\ opponent\ P_l] = Pr[Y < 1]$. If we consider the k independent trials to be Poisson trials with probability of failure per trial $p_i < a$, then by the Chernoff Bounds ([MR95]) we have:

$$\begin{aligned} Pr[Y < (1-\delta)\mu] &< \exp\left(-\frac{\mu\delta^2}{2}\right),\ for\ any\ \delta > 0. \\ (1-\delta)\mu &= 1 \Rightarrow \delta = 1 - \frac{1}{\mu} \end{aligned}$$

where $\mu = E[Y] = (1-a)k$ (by linearity of expectation). Thus, the following holds:

$$\begin{aligned} Pr[Y < 1] &< \exp\left(-\frac{\mu(1-\frac{1}{\mu})^2}{2}\right) \\ &= \exp\left(-\frac{(\mu-1)^2}{2\mu}\right) \end{aligned}$$

The probability of a specific new subtree failing to be constructed by the duel of master processors P_j and P_l will be:

$$\begin{aligned} Pr[subtree\ fails] &= Pr[P_j's\ Y < 1] \cdot Pr[P_l's\ Y < 1] \\ &< \exp\left(-\frac{(\mu-1)^2}{\mu}\right) \end{aligned}$$

and the probability of a failure occurring in level i that has $\frac{n}{2^i}$ nodes, is

$$Pr[level\ i\ fails] < \frac{n}{2^i}\exp\left(-\frac{(\mu-1)^2}{\mu}\right)$$

Finally, the probability of the whole binary tree failing to be constructed properly will be:

$$\begin{aligned} P_{\text{fail}} &= \sum_{i=\log\log n+1}^{\lceil \log n \rceil} Pr[level\ i\ fails] \\ &< \sum_{i=\log\log n+1}^{\log n+1} \frac{n}{2^i} \cdot \exp\left(-\frac{(\mu-1)^2}{\mu}\right) \\ &< n \cdot \exp\left(-\frac{(\mu-1)^2}{\mu}\right) \cdot \sum_{i=\log\log n+1}^{\log n+1} \frac{1}{2^i} \end{aligned}$$

But,

$$\begin{aligned} \sum_{i=\log\log n+1}^{\log n+1} \frac{1}{2^i} &= \sum_{i=1}^{\log n+1} \frac{1}{2^i} - \sum_{i=1}^{\log\log n} \frac{1}{2^i} \\ &= \left(2 - \frac{1}{2n}\right) - \left(2 - \frac{1}{2\log n}\right) \\ &= \frac{1}{2}\left(\frac{1}{\log n} - \frac{1}{n}\right) \qquad (1) \end{aligned}$$

Thus we have

$$\begin{aligned} &< n \cdot \exp\left(-\frac{(\mu-1)^2}{\mu}\right) \frac{1}{2}\left(\frac{1}{\log n} - \frac{1}{n}\right) \\ &< \frac{1}{2}\left(\frac{n}{\log n} - 1\right) \cdot \exp(-\mu + 2) \\ &= \frac{1}{2}\left(\frac{n}{\log n} - 1\right) \cdot 2^{-\frac{\mu-2}{\ln 2}} \qquad (2) \end{aligned}$$

If we set the number of trials during the search of the opponent for a duel to be $k = c \cdot \log n$ then $\mu = (1-a) \cdot k = (1-a) \cdot c \cdot \log n$ and

$$\begin{aligned} P_{\text{fail}} &< \frac{1}{2}\left(\frac{n}{\log n} - 1\right) \cdot 2^{-\frac{(1-a)c\log n - 2}{\ln 2}} \qquad (3) \\ &< \frac{1}{2}\left(\frac{n}{\log n} - 1\right) \cdot n^{-\frac{(1-a)c-1}{\ln 2}} = \mathcal{O}(n^{-c_1}) \end{aligned}$$

where

$$\begin{aligned} c_1 &= \frac{(1-a)c-1}{\ln 2} - 1 = \\ &= \frac{1-a}{\ln 2} \cdot c - \left(1 + \frac{1}{\ln 2}\right) \end{aligned}$$

■

Proof of Lemma 4.1

Consider a specific message s sent by a processor P_i to a processor P_j in $\mathcal{VM}$. Then, in $\mathcal{RM}$, for all pairs of active processors $Q_{i,k}$, $Q_{j,k}$, $k = 1, \cdots, c\log n$, s is transmitted from $Q_{i,k}$ to $Q_{j,k}$. The probability that s fails to be transmitted through any pair of processors, is equal to the probability that for all pairs, at

least one of the two processors is inactive. This probability is $< (2a)^{c \log n}$, where $a < 1/2$. Since we have at most n groups and at most $(n-1)$-relations to be implemented, the statement of the lemma holds with probability $1 - n^{c'}$, where c' is a constant, for appropriately chosen c. ■