Theory Comput. Systems **33**, 427–464 (2000)
DOI: 10.1007/s002240010009

Theory of Computing Systems

Robust Parallel Computations through Randomization*

S. C. Kontogiannis,[1,2] G. E. Pantziou,[2,3] P. G. Spirakis,[1,2] and M. Yung[4]

[1]Computer Engineering and Informatics Department,
Patras University,
GR-26500 Rion, Patras, Greece

[2]Computer Technology Institute,
Riga Feraioy 61, GR-26221 Patras, Greece
{kontag,pantziou,spirakis}@cti.gr

[3]Computer Science Department,
Technological Educational Institute of Athens,
GR-12210 Athens, Greece

[4]Certco, 55 Broad Street, 22nd suite,
New York, NY 10004, USA
moti@cs.columbia.edu; moti@certco.com.

Abstract. In this paper we present an efficient general simulation strategy for computations designed for fully operational BSP machines of n ideal processors, on n-processor dynamic-fault-prone BSP machines. The fault occurrences are fail-stop and fully dynamic, i.e., they are allowed to happen on-line at any point of the computation, subject to the constraint that the total number of faulty processors may never exceed a known fraction. The computational paradigm can be exploited for robust computations over virtual parallel settings with a volatile underlying infrastructure, such as a NETWORK OF WORKSTATIONS (where workstations may be taken out of the virtual parallel machine by their owner).

Our simulation strategy is Las Vegas (i.e., it may never fail, due to backtracking operations to robustly stored instances of the computation, in case of locally unrecoverable situations). It adopts an adaptive balancing scheme of the workload among the currently live processors of the BSP machine.

Our strategy is efficient in the sense that, compared with an optimal off-line adversarial computation under the same sequence of fault occurrences, it achieves

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

an $\mathcal{O}\big((\log n \cdot \log\log n)^2\big)$ multiplicative factor times the optimal work (namely, this measure is in the sense of the "competitive ratio" of on-line analysis). In addition, our scheme is modular, integrated, and considers many implementation points.

We comment that, to our knowledge, no previous work on robust parallel computations has considered fully dynamic faults in the BSP model, or in general distributed memory systems. Furthermore, this is the first time an efficient Las Vegas simulation in this area is achieved.

1. Introduction

The issue of fault tolerance in the framework of Parallel and Distributed Computing[1] tries to capture phenomena where some nodes (or communication links) of a target decentralized machine corrupt during the execution of a parallel algorithm. The issue has become very intriguing in recent years, due to the demand for execution of parallel algorithms over arbitrary sets of machines that work as a whole. In addition, the necessity of exploiting off-the-shelf computational power has lead to the consideration of arbitrary environments of Decentralized Computing that may vary with time, according to the availability of their building blocks. Thus, it would be very interesting to devise techniques that use an environment which is prone to failures, for the emulation of a similar environment which is guaranteed to be fault-free during the execution of a parallel algorithm. Fault tolerance in the context of Decentralized Computing can be provided at various levels of such an unstable computing environment:

- At the **machine level**, where the underlying environment is actually a fixed network of processing elements that tries to overcome the corruption of a specific node (or edge). The works of this kind are usually based on two major techniques. The first is the technique of embedding an ideal parallel machine into a fixed, fault-tolerant underlying network of processing elements which can be fault-tolerant with only constant slowdown (e.g., in [29] it is shown that an n-node butterfly or shuffle-exchange network can emulate a fault-free network of the same type and size, with only constant slowdown). The second technique for providing fault tolerance at machine level, is the technique of redundant computations. Albeit it seems that redundancy in parallel computations is rather a waste of computational power, this technique is as powerful as randomization in some cases, especially when we have to deal with static faults (see in Section 1.1 the categorization of fault occurrences).
- At the **cost model level**, where the abstract machine model that is considered by the programmer, is itself fault-tolerant. In particular, the cost model tries to exploit the underlying realistic machine in such a way that the overall execution of the input algorithm will not be affected by the corruption of arbitrary processing elements at runtime. The major difference between this and the previous category of fault tolerance is that, in the latter category, the augmented cost models may

[1] In what follows we call the area of Parallel and Distributed Computing **Decentralized Computing**. This terminology is also used in [24]).

become the middle-tier technologies between the actual (arbitrary) environments that provide parallel computing capabilities, and the programming environments that need to consider general-purpose parallel architectures in order to implement parallel algorithms transferable from parallel machine to parallel machine. Most works of this kind focus on the PRAM cost model, which used to be the most popular model of Parallel Computations until the early nineties (e.g., [22], [23], [26], [25], [7], [8]). Recently some new works on fault-tolerant versions of more realistic cost models than PRAM (such as BSP) have arisen, which seem to provide general solutions for fault-tolerant versions of these realistic cost models (this work, along with [27], is in this flavor).

- At the **programming environment level**, in which the programming environment itself takes over the responsibility of providing fault-tolerant primitives to the designers of parallel algorithms (e.g., synchronization operations, end-to-end guaranteed communication, robustness of the storage scheme, agreement protocols among the live processors, etc.). Some of these works are [16], [12], [11], and [15].

1.1. *Categorization of Faults*

The major distinction of the fault-tolerance problems is based on the kind of fault occurrences they consider. In the area of Parallel Computing, the prevailing model of faults is the **fail-stop** model, introduced by Kanellakis and Shvartsman [23], according to which whenever a processor dies it is excluded from the remaining simulation process. In [22] the processing elements are allowed to restart at arbitrary times (this is the so-called **restartable fail-stop** model). Of course, in that case serious problems with the coordination of work might arise, which are usually dealt with by definite synchronization operations, or timestamping techniques.

Similarly, in the area of Distributed Computing we may have to cope with a large range of faults, from **crash** faults (which are equivalent to the fail-stop model) to **omission** faults, or even **malicious** or **byzantine** faults (where the faulty processors join forces to affect the rest of the simulation process). In most cases the crash faults case is considered, while the malicious faults case has to do mainly with issues such as virus attacks in decentralized computing environments, secure storage, etc.

Additionally the faults may be classified as **static** (i.e., known at the beginning of the simulation process) or **dynamic** (i.e., they may occur at arbitrary points during the simulation process). Both these cases are quite interesting. More specifically, the static case reflects the adaptation of a certain cost model or parallel computing environment, over an unknown (but fixed from that point on) working environment. On the other hand, the problem of tolerating dynamic processor faults in pragmatic settings, can be seen as a {**Safe Storage & Checkpoint & (dynamic) Scheduling**} problem. The challenge lies in proposing an efficient strategy that will achieve an almost work-preserving, robust execution of the input algorithm and will also assure a balanced split of the workload among the operational processors. This strategy will also have to exploit a robust storage scheme that will tolerate arbitrary processor failures and will also provide a (periodic) checkpoint procedure to commit work of the simulation process at runtime. Typical examples of this approach are [11] and the present work.

1.2. *Related Work*

As already mentioned, the issue of fault-tolerant computations in decentralized computing environments covers a wide range, from PRAM machines to arbitrary decentralized computing environments. In the following sections we present a synopsis of the most important articles in which fault tolerance has been dealt with in the literature of Decentralized Computing.

1.2.1. *Fault Tolerance on* PRAM *Machines*. There have been many works in the area of fault tolerance on the PRAM cost model, especially in early nineties [25], [26], [22], [23], [9]. In these works the fail-stop model is adopted. The reason was that the PRAM model considers that the input algorithms are executed in a lock-step fashion, and the processing elements are totally synchronized by a global clock. Thus, there is no chance of delayed action of a processing element that might completely mislead the whole computation. On the other hand, the malicious (or byzantine) faults necessitate the coordination of work among the live processors via the robust implementation of agreement protocols, which are unnecessary in the PRAM case due to the completely synchronous operation of the processing elements and the (robust) Shared Memory feature.

In [26] a general strategy for simulating arbitrary CRCW PRAM steps over a setting that allows dynamic processor faults is provided, by solving a core problem for this cost model, the `Certified Write All` problem. Note that the existence of simulations of arbitrary PRIORITY CRCW PRAM steps (which is the strongest PRAM variant) by at most $\mathcal{O}(\log n)$ EREW PRAM steps, implies corresponding results for the weakest variant of this cost model as well. In [25] the same problem is dealt with over the restartable fail-stop model, by using a combination of tentative computations (i.e., computations that are most likely to be correct) and definite computations (guaranteed computations against any sequence of fault occurrences). This approach achieves constant amortized slowdown per CRCW PRAM step for many reasonable fault distributions. This is done by having the processors tentatively simulate the fault-free execution of the input algorithm, while a definite auditing procedure monitors the simulation process at specific points. In [23] and [22] some strategies are provided for dealing with the `Write All` problem in the fail-stop, nonrestartable/restartable cases, where the faults may occur dynamically during the execution of an input CRCW PRAM algorithm.

The reader is referred to the monograph of Kanellakis and Shvartsman [4] for an overview of the most important simulation strategies that deal with the issue of fault tolerance on PRAM machines and an excellent classification of the instances of fault tolerance on PRAM machines in the fail-stop model.

1.2.2. *Fault Tolerance on Arbitrary Machines*. In the case of arbitrary computing environments, that consist of processing elements communicating via an underlying network infrastructure, there are several crucial parameters other than the computational power provided by the machine that affect the performance of fault-tolerant strategies. Such parameters are for example the latency of the communication infrastructure, the bandwidth per processor, and the synchronization cost, that are not accounted for in the PRAM cost

model. Thus, proprietary solutions for providing fault tolerance in such settings should be provided, or more appropriate cost models should be chosen, that are closer to the actual overhead of a simulation strategy.

In [12] the case of dynamic processor faults is considered over an arbitrary message-passing underlying computing environment of synchronous machines. In this setting, an optimal strategy is provided for executing a set of independent tasks. In this work it is stated that the core of any simulation strategy over a synchronous message-passing environment is a CHECKPOINT routine of the remaining live processors, along with a BALANCED ALLOCATION strategy. In [16] another primitive operation of distributed computing is considered, namely, the BYZANTINE AGREEMENT. In this work a BYZANTINE AGREEMENT protocol is provided that is robust against crash failures and has optimal message complexity. Then it is used as a primitive operation for the provision of a family of early stopping agreement with improved message complexity and a new solution to the CHECKPOINTING procedure that was provided in [12].

Another work that deals with matters of synchronization over computing environments of limited asynchrony is [17], where a general strategy for simulating a completely synchronous network of processing elements (such as a PRAM machine) by a network of limited asynchrony is provided. Despite the fact that this work considers a totally reliable underlying network, it is interesting that it uses the notion of tentative computations and definite auditing (or checkpointing) procedures for safe progress of the simulation process, which is a strategy that was exploited in many works of fault tolerance, including the current work.

Although such proprietary approaches achieve great efficiency (and in some cases optimality) in the general case of an arbitrary synchronous message-passing computing environment, they cannot exploit the feature of bulk synchrony provided by some new cost models (e.g., BSP, QSM, CGM) that seem to prevail in the area of Parallel Computation in the last few years. It should also be noted that bulk synchrony (or limited asynchrony) is inherent in the parallel algorithms themselves in many cases, and this gives rise to the provision of bulk synchronous, fault-tolerant environments, that focus their power on features other than the continuous synchronization and agreement protocols, such as Load Balancing and Robust Storage Schemes.

1.2.3. *Fault Tolerance on* NETWORKS OF WORKSTATIONS. The NETWORKS OF WORKSTATIONS [11], [3] platform tries to satisfy the demand for the construction of parallel systems using off-the-shelf workstations that deliver and in many cases even surpass the power and reliability of many large-scale machines. It actually represents an inherently asynchronous (or bulk-synchronous) environment for the execution of parallel algorithms. It seems that a cost model such as QSM or BSP would be very easily applicable to this parallel setting, because it consists of processing elements communicating via a specific communication infrastructure, and operates in an asynchronous or bulk-synchronous mode. In the work of [11] the NETWORK OF WORKSTATIONS is modeled as a completely asynchronous multiprocessor, shared-memory system, augmented with a fault-tolerant mechanism that treats even the slow workstations as failed ones. The architecture of this system is centralized, in the sense that there are specialized processors that perform specific operations (i.e., there is one task manager that schedules the pending work, some processors that deposit the necessary information, and some work-

ers that actually execute the tasks that are assigned to them). Nevertheless, as stated in the present work, an optimal fault-tolerant strategy should minimize job migration and should be integrated into the parallel system itself. This work actually tries to exploit the techniques that have appeared in the literature for providing fault tolerance over PRAM, combined with robust storage schemes based on information dispersal techniques (see [32]). Of course fault-tolerant strategies on more relevant cost models will be much more realistic.

1.2.4. *Fault Tolerance on* BSP *Machines*. The BSP cost model focuses mainly on the {computation & communication & bulk synchrony} cost during the execution of an algorithm, rather than on the continuous synchronization procedure of a completely asynchronous setting, as in the cases of [16] and [12]. Thus new strategies are necessitated that exploit this special characteristic of bulk synchrony and will provide fault tolerance on BSP machines.

In [27] the issue of fault tolerance over BSP machines has been addressed. Simulations for two different cases were considered. In the static case, the faulty or unavailable processors are already known at the beginning of the computation and no processor changes its status afterwards. On the other hand, in the (semi)dynamic case, each processor may fail or become unavailable with a fixed probability during the computation and remains so until the end of the computation; however, some critical periods during the computation where no processor was allowed to fail, could not be avoided. In this work, some Monte Carlo constructions based on embedding of the virtual BSP machine on the operational subset of the real BSP machine (for the static case) and of work redundancy (for the semidynamic case) assure efficient executions of BSP algorithms over fault-prone BSP machines.

1.3. *Our Contribution*

In this paper we generalize the work of [27] and consider fault-tolerant BSP computations under fully dynamic processor faults without assuming any fault-free periods.[2] Namely, the faults may happen on-line at any point of the computation. To tackle the problem, the issue of the fault tolerance on BSP is modeled as an independent-jobs scheduling problem, on a dynamically changing computing environment. To be more specific, consider having an algorithm that is designed for an ideal, fault-free n-processor BSP machine. Each virtual superstep of this algorithm may be thought of as a set of n independent computational threads that impose some communication demands (i.e., the implementation of an h-relation among these threads), and correspond to the work to be done by each virtual processor during the current virtual superstep. Our task is to assign this amount of work on a dynamically changing set of live (or simply available, or not stalled) processors, in such a way that, as long as there are at least $(1 - a)n$ live processors (a is an input parameter of the realistic setting that will be used), this amount of work will be successfully executed. The goal is to choose an efficient strategy that will achieve an almost work-preserving, robust execution of the BSP algorithm, and will also assure a balanced split of the workload among the operational processors. We propose a modular and efficient

[2] A preliminary version of the current work was presented in SPAA 98 [28].

simulation scheme which, compared with an optimal off-line adversarial computation under the same sequence of fault occurrences, achieves an $\mathcal{O}\big((\log n \cdot \log\log n)^2\big)$-factor times the optimal work.

The proposed scheme is Las Vegas, i.e., it always completes the computation successfully. This is so, due to a BACKTRACKING process, which retrieves robustly stored instances of the simulation process in case an interruption to the flow of the computation has occurred, due to locally unrecoverable situations.

The proposed strategy combines an ADAPTIVE LOAD BALANCING scheme with a MIXED STORAGE scheme (based on Rabin's *Information Dispersal Algorithm* [32]) and a CHECKPOINTING procedure (that exploits a BSPAGREEMENT Protocol for periodic synchronization among the still live processing elements).

In what follows we present the Robust-$\mathcal{BSP}$ simulation strategy for handling processor failures on the BSP cost model. This strategy has to face fully dynamic processor faults and a more complex approach is adopted, that combines a balancing scheme, a storage scheme, and a checkpointing procedure. In Section 2 we give a brief description of the BSP cost model, and we introduce the `Sparse Occupancy` problem, that will be used in our analysis. In Section 3 we present the major routines used by Robust-$\mathcal{BSP}$ and we present our mixed storage scheme that assures the robustness of the whole process. In Section 4 we present the adaptive balancing scheme that has been adopted by our simulation strategy.

2. Preliminaries

2.1. *The* BSP *Cost Model*

The **Bulk Synchronous Parallel** (BSP) model was introduced by Valiant [34] as a bridging model that tries to close the gap between the domains of decentralized architectures and parallel algorithms.

The applicability of the BSP cost model lies in the fact that, apart from the cost of the parallelism that is accounted for by the traditional PRAM cost model, it also considers the communication and synchronization limitations that are imposed by the realistic decentralized architectures. Yet, it does not limit the interoperability of the model among different decentralized computing environments, by abstracting away from the designers detailed architectural features such as the topology of the processing elements, or the synchronization procedures. Thus, the objective of this model is to allow the design of parallel algorithms that can be efficiently executed on a variety of decentralized architectures, at a predictable cost, with respect to some architectural parameters that reflect the capabilities of the underlying decentralized machine.

A BSP algorithm $\mathcal{A}$ consists of a sequence of **supersteps** that are separated by **Bulk Synchronization** operations (SYNC in short). Each superstep consists of a **Local Computation** phase and a **Communication** Phase (LC-phase and Comm-phase, respectively). During the LC-phase of a superstep each processor performs a sequence of operations on data held in its local memory, while the Comm-phase takes over the transmission of the outgoing messages of each processing element to their destinations,

via the underlying communication infrastructure. At the end of the superstep a SYNC operation indicates the end of the current superstep. A BSP machine consists of the following components:

- A collection of n identical **processor/memory elements** which are distinguished by their unique **identification numbers**.
- A **communication infrastructure** takes over the point-to-point communication process. This infrastructure is characterized by the bandwidth g and the latency parameter L which are explained in the next paragraph).
- A **barrier synchronization mechanism** among the n processing elements.

The two parameters of the decentralized architecture (apart from the number n of processing elements) that are taken into account by BSP are the **bandwidth** g, i.e., the (per-processor) ratio of the total throughput of the whole system in terms of local computation operations, to the throughput of the underlying communication network in terms of words of information delivered, and the **latency** L, which is the minimum time interval between two consecutive SYNC operations.[3] Thus, the running time of a single superstep on the BSP cost model is characterized by the parameters n, g, and L, and is given by the following formula:

$$T_{\text{superstep}} = \max\{L, T_{\text{LC}} + T_{\text{Comm}}\}, \tag{1}$$

where T_{LC} is the maximum (among the processing elements) cost for local computations, and T_{Comm} is the maximum time needed for transmitting all the outgoing messages to their destinations. If we consider that during the Comm-phase each processor sends and receives at most h one-word messages (i.e., an h-relation has to be implemented during the Comm-phase), then $T_{\text{Comm}} = g \cdot h$, according to McColl [30]. In that case, we have

$$T_{\text{superstep}} = \max\{L, T_{\text{LC}} + g \cdot h\}. \tag{2}$$

Remark. In some cases the underlying machine charges the implementation of an h-relation as $g \cdot \max\{h, h_0\}$ for some h_0 that depends on the machine. This is due to a fixed communication initialization cost, which is irrelevant to the size of the h-relation to be implemented by the network infrastructure.

Finally, the following fact will be used in our time estimations in what follows (for a justification of this fact, the reader is referred to [18] and [4]):

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

2.2. *The Sparse Occupancy Problem*

In this subsection we present some basic `Balls&Bins` problems that will be useful in the analysis of our simulation strategy in Section 3.

[3] Observe that L is a lower bound on the duration of a single superstep.

Assume we have a number of M independent sources that throw balls independently and uniformly at random into n identical bins. We study the case of each source throwing $m \ll n$ balls into the n bins. We call this family of problems the family of `Sparse Occupancy` problems.

Lemma 2.1. *Consider having a single source that throws $m \ll n$ balls into the n bins. The probability of each bin being occupied by at most one ball tends to unity.*

Proof. Consider a specific bin (e.g., $\mathcal{B}_1$) and let

$$
\begin{array}{ll}
p_i \equiv \mathbb{P}[\mathcal{B}_1 \text{ has exactly } i \text{ balls}], & \\
\check{p}_i \equiv \mathbb{P}[\mathcal{B}_1 \text{ has at least } i \text{ balls}], & \hat{p}_i \equiv \mathbb{P}[\mathcal{B}_1 \text{has at most } i \text{ balls}], \\
\check{P}_i \equiv \mathbb{P}[\exists \text{ bin with at least } i \text{ balls}], & \hat{P}_i \equiv \mathbb{P}[\text{Every bin contains at most } i \text{ balls}].
\end{array}
$$

We seek $\hat{P}_1 = 1 - \check{P}_2$. Observe that $\check{P}_2 \leq \sum_{i=1}^{n} \check{p}_2 = n \cdot \check{p}_2$ and $\check{p}_2 = 1 - \hat{p}_1$. However,

$$
\begin{aligned}
\hat{p}_1 = p_0 + p_1 &= \left(1 - \frac{1}{n}\right)^m + \binom{n}{1} \cdot \left(1 - \frac{1}{n}\right)^{m-1} \cdot \frac{1}{n} \\
&= 2\left(1 - \frac{1}{n}\right)^{m-1} \cdot \left(1 - \frac{1}{2n}\right) \sim 2\,\exp\left(-\frac{2m-1}{2n}\right).
\end{aligned}
$$

Hence, $\check{p}_2$ and $\check{P}_2$ are now given by

$$
\begin{aligned}
&\check{p}_2 = 1 - 2\exp\left(-\frac{2m-1}{2n}\right) \leq 2 \cdot \left(1 - \exp\left(-\frac{2m-1}{2n}\right)\right) \quad \text{and} \\
&\check{P}_2 \leq 2n \cdot \left(1 - \exp\left(-\frac{2m-1}{2n}\right)\right).
\end{aligned}
$$

Finally, we have $\hat{P}_1 \geq 1 - 2n \cdot (1 - \exp(-(2m-1)/2n)) \longrightarrow 1$, for $m \ll n$ and $n \to \infty$. □

Remark. The use of this analysis instead of using one of the classical probabilistic techniques for the Occupancy Problem (e.g., in [31] and [21] where a martingale is used to provide some tight bounds) is that these tools are based on Azuma's inequality, which is not such a strong tool to capture so small discrepancies from the expected value, as in our vase. For example, in [21] the bounds hold for $m \gg n$ in the proof with the martingales, while for $m \approx n$ there is another proof of (less tight) bounds, based on Markov's inequality.

Lemma 2.2. *Assume we have $A(n) = \mathcal{O}(n)$ sources that throw m balls each independently and uniformly at random (u.a.r. in short) into the n bins, where $m \ll n$. Then the probability of there existing a bin whose number of balls diverges from $\mathcal{O}\log n$ is polynomially small.*

Proof. Let $M(n) \equiv A(n) \cdot m$ denote the total number of balls thrown into the n bins by the $A(n)$ sources. Let also $X_{i,j}$ be the indicator variable of the jth ball going into bin i, while $B_i \equiv \sum_{j=1}^{M(n)} X_{i,j}$ is the total number of balls that go into the ith bin. Note that, $\forall i \in [n]$, $\{X_{i,j} : j \in M(n)\}$ is a set of independent, identically distributed (i.i.d. in short) random variables, while $\{B_1, \ldots, B_n\}$ are **negatively associated** random variables (for the definition of the negative association property of a set of random variables, see [14]). Note also that $\mathbb{E}[B_i] = \sum_{j=1}^{M(n)} \mathbb{E}[X_{i,j}] = 1/n \cdot A(n) \cdot m = A(n)/n \cdot m$.

Fix a bin $i \in [n]$. Then, since B_i is the sum of $M(n)$ i.i.d. random variables with success probability $1/n$, by applying Chernoff Bounds we get the following, for all $0 < \delta_1, \delta_2 \leq 1$:

$$\mathbb{P}[B_i > k_1 \log n] < \exp\left(-\frac{\mathbb{E}[B_i] \cdot \delta_1^2}{3}\right) = \exp\left(-\frac{k_1 n (\log n - 1)^2}{3A(n) \cdot m}\right),$$

$$\begin{aligned}\mathbb{P}[B_i < k_2 \log n] &< \exp\left(-\frac{\mathbb{E}[B_i] \cdot \delta_2^2}{2}\right) \\ &= \exp\left(-\frac{A(n) \cdot m}{2n} - k_2 \log n (k_2 n \log n - 2)\right),\end{aligned}$$

where

$$\delta_1 \equiv \frac{n(k_1 \log n - 1)}{A(n) \cdot m} \quad \Rightarrow \quad m \geq \frac{n(k_2 \log n - 1)}{A(n)}$$

and

$$\delta_2 \equiv 1 - \frac{k_2 n \log n}{A(n) \cdot m} \quad \Rightarrow \quad m > \frac{k_2 n \log n}{A(n)}.$$

From these last inequalities we can clearly deduce that

$$\begin{aligned}\mathbb{P}[\exists i \in [n], B_i > k_1 \log n] &< \sum_{j \in [n]} \exp\left(-\frac{k_1 n (\log n - 1)^2}{3A(n) \cdot m}\right) \\ &\leq n \cdot \exp\left(\frac{k_1 n (\log n - 1)^2}{3A(n) \cdot m}\right) = n^{-k_3}, \qquad (3)\end{aligned}$$

$$\begin{aligned}\mathbb{P}[\exists i \in [n], B_i < k_2 \log n] &< \sum_{j \in [n]} \exp\left(-\frac{A(n) \cdot m}{2n} - k_2 \log n (k_2 \log n - 2)\right) \\ &\leq n^{-k_4}, \qquad (4)\end{aligned}$$

where

$$k_3 = \frac{k_1 \cdot n (\log n - 1)^2}{3 \ln n \cdot A(n) \cdot m} - 1 \quad \text{and} \quad k_4 = \frac{A(n) \cdot m}{2n \ln n} + \frac{k_2 \log n}{\ln n} \cdot (k_2 n \log n - 1) - 1. \quad \square$$

3. A Paradigm for Robust Computations in Dynamically Changing Environments: Robust-$\mathcal{BSP}$

In this section we present the major techniques that are exploited by Robust-$\mathcal{BSP}$ in order to provide a safe computational framework for BSP algorithms over arbitrarily changing computing environments.

3.1. *The Dynamic Simulation Model*

Suppose that we are given a BSP algorithm $\mathcal{A}$, which is designed to run over a hypothetical, fault-free n-processor BSP machine (the **Virtual Machine**, $\mathcal{VM}$). Because of its BSP nature, $\mathcal{A}$ assigns atomic workloads and imposes some demands for communication (e.g., the implementation of an h-relation) among the virtual processing elements, in each superstep of $\mathcal{VM}$. We denote the atomic workload of a specific virtual processor Q_i, along with the portion of communication on behalf of Q_i for the current virtual superstep, as **thread** T_i. In that case, the execution of each virtual superstep can be considered as a balanced, multithreaded computation, while the execution of $\mathcal{A}$ may be considered as a sequence of multithreaded computations, distinguished by periodic barrier synchronization (SYNC) operations among the processing elements (see Figure 1).

Suppose also that we are given an n-processor BSP machine which is prone to processor failures (the **Real Machine**, $\mathcal{RM}$). Then the task of $\mathcal{RM}$ will be to assure the progress of the simulation process until the end of the execution of the input algorithm $\mathcal{A}$, but also to keep the total work evenly balanced among the remaining live processors and assure recovery from any sequence of faults at any time. Notice that we do not exploit any slackness in our setting, which might indicate an "optimal" use of a subset of $\mathcal{RM}$ for the improvement of our results. This is because we want to have a realistic measure of the performance of our simulation process.

Let the (uniquely) assigned workload of a real processing element P_i of $\mathcal{RM}$ comprise its PRIMARYJOB. This means that the major task of P_i in each virtual superstep will be to simulate the execution of the threads which correspond to the virtual processors

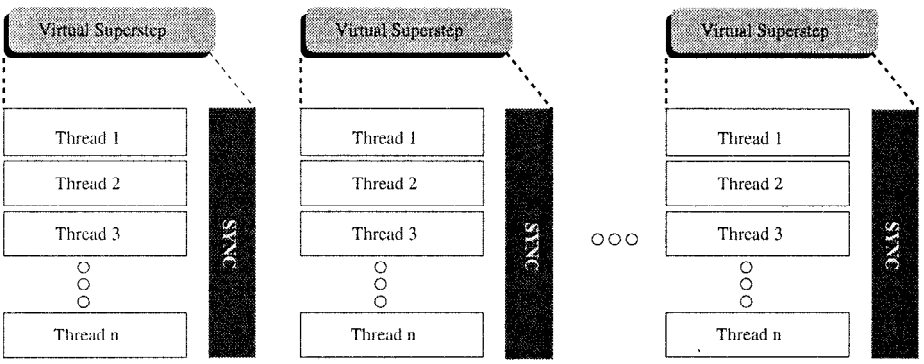

Fig. 1. Representation of a BSP algorithm as a multithreaded decentralized computation.

residing in its PrimaryJobQueue. Any additional workload that may be taken over by P_i (at its own initiative) during the simulation process, will comprise a SECONDARYJOB. This load is not necessarily uniquely assigned to the specific processor.

Initially we assume that there is a "1–1" correspondence between the processing elements of $\mathcal{RM}$ and $\mathcal{VM}$. This means that any thread of virtual processor Q_i is initially considered to be executed by the real processor P_i (which is its physical destination), unless P_i dies, in which case Q_i will have to migrate to another real processor that will have to execute its forthcoming threads. The necessary information for executing a specific thread T_m is stored by a LOCALSTORAGE scheme among the currently live real processors. This information is stored using as a key the **Virtual Identification number** ($\mathcal{V}id$) of the corresponding virtual processor. This storage scheme is based on a BSP implementation of the well know *Information Dispersal Algorithm* (IDA [32]) and assures robustness against small bursts of processor failures.

Additionally, a GLOBALSTORAGE scheme has been adopted, for the periodic creation of some robust replicas of specific instances of $\mathcal{RM}$'s status during the simulation process (we call these securely stored instances SafeStates). The SafeStates play the role of reference points to which our simulation process will backtrack, in case of locally unrecoverable situations.

We also assume that processors may share a seed from a strong pseudorandom number generator (i.e., we have "coordinated randomness" among the operational nodes of $\mathcal{RM}$).

Definition 3.1. A thread is called **assigned** if it is included in a live real processor's PrimaryJobQueue, otherwise it is called **pending**. Additionally, if a real processor has already completed the simulation of a specific thread (i.e., it has simulated its **LC-phase** and its **Comm-phase** and has safely stored the new status of the corresponding virtual processor), then this thread is considered to be **completed** till the end of the current virtual superstep. Finally, a thread that corresponds to a virtual processor Q_i and is assigned to a real processor P_j with $i \neq j$ (i.e., P_j is not the physical destination of Q_i) is called **migrated**.

For the reader's easier understanding of the measured quantities, we adopt the following terminology:

- A_k: the number of currently live processors at the beginning of virtual superstep VS_k (estimation).
- $\mathcal{C}_i/\mathcal{P}_i/\mathcal{A}_i$: the number of completed/pending/assigned threads at the end of SECONDARYJOB J_i of VS_k.
- h: the size of the h-relation that has to be implemented among the n threads during VS_k on $\mathcal{VM}$.
- F_k: the number of faults that occur during VS_k.
- $[n]$: the set $\{1, \ldots, n\}$.

For the representation of the CurrentStatus of the Virtual BSP Machine ($\mathcal{VM}$), we use the following data structures, which are robustly stored among the live processors of $\mathcal{RM}$, using a nontrivial combination of a tentative storage scheme (the LOCALSTORAGE) and a definite one (the GLOBALSTORAGE) (the reader is referred to Section 3.5.3, for the

description of the mixed storage scheme adopted):

- VLM_i: the contents of the Local Memory used by the virtual processing element Q_i. The size of the VLMs. is an input parameter for our simulation strategy. A reasonable assumption might be to consider a polylogarithmic size (e.g., $\mathcal{O}(\log^2 n)$) for these local memories.
- LoU_i: a list of still undelivered outgoing messages, on behalf of the threads hosted by the live processor P_i.
- PJ_i: the PrimaryJobQueue of P_i, containing the (uniquely assigned) threads that it executes during each PRIMARYJOB.
- **b**: the number of threads chosen to be executed by a live processor during a SECONDARYJOB.
- N_u: the size of a local neighborhood of real processors after u BACKTRACK operations have occurred, with $\mathsf{N}_0 = \log n$ and $\mathsf{N}_{u-1} = 2^u \cdot \log n$.

3.2. *Assumptions*

Regarding the model of faults, we consider that an oblivious adversary determines a sequence of processor failures that are randomly distributed all over the whole simulation of a specific input algorithm $\mathcal{A}$, and these faults are unrecoverable (i.e., we adopt the nonrestartable fail-stop model, where the dynamic faults are distributed uniformly during the whole simulation process).

This fault model tries to capture the behavior of a decentralized setting, such as a NETWORK OF WORKSTATIONS, or an arbitrary distributed computing environment, that behaves like a dynamically changing parallel machine through a virtual parallel interface (e.g., MPI or PVM). This dynamic computing environment initially allocates processors to the simulation process according to the demands of the input algorithm, and then reclaims resources (according to the demands of the whole parallel setting) because of processor unavailability.

It is also assumed that the amount of processor failures during the simulation process is upper-bounded by a fraction a, which is an input parameter to our simulation strategy. Clearly, this bound affects the cost for creating SafeStates during the simulation process, to which the strategy will backtrack, when some unexpected behavior of the parallel setting occurs, without jeopardizing the whole simulation process.

From the above two restrictions (i.e., the random distribution of the faults during the simulation process and the existence of an overall upper-bound on the fraction of faults), it becomes apparent that if we divide the simulation time into sufficiently large (e.g., polylog(n)) time intervals, there is an upper bound $r = r(a)$ on the fraction of faults that may occur in each of these intervals and it is easily shown that the concentration around this bound is very sharp (this is easily understood by a simple application of the well know `Coupon Collector's Problem`). This implies that if a virtual superstep VS_k needs polylog(n) time to be executed, then $F_k \leq rn$.

For the communication part, we suppose that the BSP algorithm $\mathcal{A}$ to be executed imposes regular h-relation implementations (for some h, which is a parameter of the input algorithm), which means that the corresponding communication graph among the n threads is an h-regular digraph. In case this is not true, some dummy messages could be used so as to have a regular communication digraph.

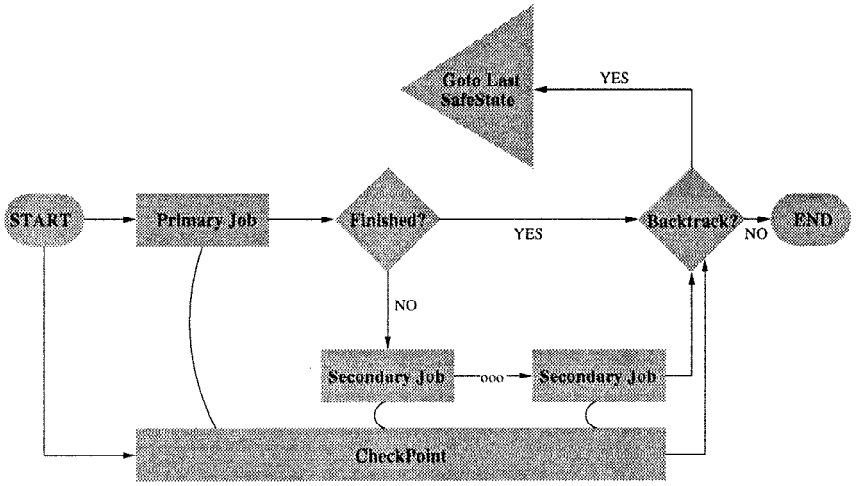

Fig. 2. The flowchart of Robust-$\mathcal{BSP}$.

3.3. *Overview of Robust-$\mathcal{BSP}$*

Robust-$\mathcal{BSP}$ considers a BSP algorithm $\mathcal{A}$ with a polynomial number of virtual supersteps, and divides it into **epochs** of K_0 virtual supersteps (the size of each epoch can be as large as $K_0 = \Theta(n)$, as will be demonstrated later by the analysis of the corruption probabilities of our strategy). It then tries to simulate robustly the execution of the next epoch of $\mathcal{A}$ and create a new SafeState at the end of it, so as to have the work done up to this point committed. In what follows we deal with the simulation of a single epoch of virtual supersteps and the presentation is done in a top-down fashion, starting from an overview of Robust-$\mathcal{BSP}$ (see Figures 2 and 3), and then we describe the major techniques that we use.

[1] **For** $k = 1$ **TO** K_0 **PARDO**
[2] Perform a PRIMARYJOB for VS_k.
[3] **IF** (NOT_FINISHED) **THEN PARDO** y times:
[3.1] **IF** CHECKPOINT == OK
[3.2] **THEN** Perform a new SECONDARYJOB
[3.3] **ELSE** Disseminate the new corruption information during the next BSPAGREEMENT
[4] **ENDPARDO**
[5] **IF** CHECKPOINT ! = OK **THEN** BACKTRACK to the last SafeState
[6] **ELSE IF** $k = K_0$ **THEN** Perform a GLOBALSTORAGE **AND** set $k := 1$
[7] **ENDFOR**

Fig. 3. The Robust-$\mathcal{BSP}$ simulation strategy.

The purpose of Robust-$\mathcal{BSP}$ is initially to let the live processors of $\mathcal{RM}$ try to execute the current virtual superstep VS_k as if there were no more processor failures and the workload were evenly distributed among them (this is done during the routine PRIMARYJOB). In case VS_k is not yet completed, the overall strategy executes y SECONDARYJOBS, so that the work of VS_k is finished and the pending workload is evenly distributed among the remaining live processors. As will be shown in what follows, a double logarithmic number of SECONDARYJOBS is enough for our strategy to work.

Remark. The PrimaryJobQueues of the remaining live processors at the end of the previous virtual superstep (VS_{k-1}) comprise a partition of the set of virtual processors into disjoint and almost evenly balanced sets of threads to be executed. Thus, provided that no more processor failures occur during the PRIMARYJOB of VS_k and the workload is already evenly balanced among the live processors of $\mathcal{RM}$, this part would be sufficient for the completion of the work of VS_k. Notice that no work replication is imposed at this part of our strategy.

To assure the robustness of our simulation against processor failures, we deploy a fast storage scheme (see Section 3.5.3), according to which the CurrentStatus of $\mathcal{VM}$ is stored during each superstep in some properly constructed neighborhoods of real processors, the size of which depends on the fault occurrences (up to the specific point of the simulation process). A GLOBALSTORAGE scheme is also deployed for creating a SafeState at the end of each epoch: that is a robust storage of a specific instance of $\mathcal{VM}$'s CurrentStatus so as to be able to reconstruct it at any time, if a neighborhood of processors has been corrupted. The LOCALSTORAGE procedure is based on an implementation of the well known *Information Dispersal Algorithm* (IDA [32]), while the GLOBALSTORAGE procedure creates the necessary replicas of LOCALSTORAGES so that up to arbitrary $a \cdot n$ processor failures can be overcome.

In the case of a new corruption of the LOCALSTORAGE having been discovered by a processor, the dissemination of this exceptional information will occur the next time an agreement protocol for coordination of work among the remaining live processors is performed. This is done at the beginning of the next job (either a PRIMARYJOB or a SECONDARYJOB). At the end of the current epoch, a BACKTRACK routine is performed by all live processors to the last SafeState, in case a locally unrecoverable error has occurred, otherwise a new SafeState is created and the work of the current epoch is then completed.

Remark. A possible new processor failure that will not let the corruption information be disseminated to all the live processors, or a failure of the GLOBALSTORAGE may cause no problem at all. This is because the problematic situation will be discovered during the next superstep, and a new BACKTRACK procedure will fix this abnormality. Additionally, as will be demonstrated in Section 3.5.2, each time a new BACKTRACK occurs, the probability of having a new corruption decreases the significantly because of the redistribution of faults in the new, double-sized neighborhoods of processors and thus there is no chance of an infinite loop in the same epoch.

In our simulation strategy, an ADAPTIVE LOAD BALANCING scheme (ALB in short) is adopted. It tries to distribute the workload of $\mathcal{VM}$ evenly among the currently operational processors of $\mathcal{RM}$. The purpose of ALB is to make the PRIMARYJOB routine work as close to optimal as possible, in the case when the parallel setting is at a stable state (i.e., when no more processors dies for a while). This is a consequence of the "almost even workloads among the operational processors" property, achieved by our balancing scheme. This is actually an on-line load-balancing technique based on ideas such as work stealing [6] and dynamic contest among the live processors for the assignment of new work. In Section 4, ALB is shown to have a very good performance with respect to our simulation model.

The "FINISHED?" condition of the Robust-$\mathcal{BSP}$ strategy can be implemented by the network infrastructure that will have to add the number of completed threads in the PrimaryJobQueues of the live processors, and will signify the end of VS_k if n is reached by this sum. This added intelligence of the network infrastructure is necessitated by the fact that apart from the periodic synchronization operation among the virtual processors is also necessary, that will signify the end of the virtual supersteps in the ideal machine ($\mathcal{VM}$). Another approach could be the use of a tentative estimation for this Boolean condition, by having the processors apply a BSPAGREEMENT protocol to decide whether the simulation of VS_k has been completed or not. Of course, a solution like this would require an assured load-balancing property (i.e., very tight upper and lower bounds on the PrimaryJobQueues) for the failure probability to be extremely small. On the other hand, some checkpointing procedures such as those proposed in [16] and [12] could be adapted to provide this virtual SYNC operation that would assure the integrity of $\mathcal{VM}$. Recall that a possible failure at this point would not be catastrophic since it would be discovered anyway during the next virtual superstep.

As for the "BACKTRACK?" condition at the end of the current virtual superstep, if a new corruption of the LOCAL STORAGE has been discovered by a processor, the dissemination of this exceptional information will occur the next time a BSPAGREEMENT protocol is executed by the remaining live processors of $\mathcal{RM}$. This is done at the beginning of the next job (either a PRIMARYJOB or a SECONDARYJOB). At the end of the current superstep, a BACKTRACK routine is performed by all the live processors to the last SafeState in case a locally unrecoverable error has occurred. Otherwise, the simulation process proceeds with a new virtual superstep or with the creation of a new SafeState and the work of the current epoch is then completed.

3.4. *The* BSPAGREEMENT *Protocol*: *a Basic Technique*

The purpose of an agreement protocol is to be able to have all live processors of the BSP machine agree in a unique value. In our BSPAGREEMENT protocol (see Figure 4), each live processor computes an **Initial Value** and subsequently an agreement rule (e.g., majority, median, etc.), which is given as an input parameter to this protocol, is employed in order to have all live processors agree on a unique value. For example, suppose that we have to make a unique estimation of the number of currently live processors or choose a unique random seed for the later choice of a hash function, common to all live processors.

[1] **Initial Values:** Each processor estimates independently an initial value.
[2] **PARDO** x times:
[2.1] **BSPAgreement:** Each processor sends $\log n$ messages to random target processors.
[2.2] For each live processor, $NewEstimation = AgreementRule(samples)$

Fig. 4. The BSPAGREEMENT protocol.

At the beginning, each live processor acquires its own **Initial Value** (e.g., RANDOMSAMPLING is applied by each live processor to get an estimation of the number of live processors in $\mathcal{RM}$ or a random seed is chosen independently by each live processor). Subsequently, some rounds of agreement attempts are performed in order for all live processors eventually to end up with the same value or with values that are very close to each other. In a single round, each live processor estimates a new value according to the agreement rule that is applied on a new random sample of values.

Remark. The agreement rule depends on the nature of the problem to be solved. For example, a "median" agreement rule for the choice of a unique random seed would destroy the randomness of a common seed, while a "majority" rule would work, provided that the **Initial Values** are truly randomly and uniformly chosen. On the other hand, if the problem is the estimation of the number of currently live processors, A_k, then a "weighted median" among the newly sampled values seems more reasonable than a "majority" rule that would have no sense in this case.

Case study: *estimation of currently live processors*. In this subsection we demonstrate how we use the BSPAGREEMENT protocol to have the remaining live processors acquire consistent estimations of their number in $\mathcal{RM}$, i.e., with their variance converging to zero.

The **Initial Values** of step [1] (see Figure 4) are determined by an application of RANDOMSAMPLING independently by each live processor: each live processor throws independently and uniformly at random (u.a.r.) $\mathcal{O}(\log n)$ polling messages (which are sent to distinct targets with overwhelming probability according to the `Sparse Occupancy` problem, see Section 2.2). Then each polled live processor responds to the polling by sending a message to the corresponding requester.

In step [2] a sequence of rounds is performed among the live processors, so that they eventually agree in tight estimations of the number of live processors in $\mathcal{RM}$.

Lemma 3.1. *The requested number of rounds of the* BSPAGREEMENT *protocol, until all processors have agreed* (*with polynomially small deviation probability*) *on a unique value, is* $\mathcal{O}(\log n/\log \mathtt{N})$, *where* `N` *is the size of the neighborhood of the* RANDOMSAMPLING *applications*.

Proof. We proceed in our analysis by following the BSPAGREEMENT routine step by step:

Step 1. Each live processor P_j samples $\mathtt{N}$ real processors from $\mathcal{RM}$. Consider one indicator variable $X_{i,j}$ for the ith sample of P_j, which is

$$X_{i,j} = \begin{cases} 1 & \text{if the } i\text{th sample of } P_j \text{ corresponds to a live processor} \\ 0 & \text{otherwise.} \end{cases}$$

RANDOMSAMPLING is actually the implementation of two $\mathcal{O}(\mathtt{N})$-relations, one during which each live processor polls $\mathtt{N}$ processors chosen u.a.r. from $\mathcal{RM}$ (with replacement), and one for getting the answers by the live polled processors. If at the beginning of VS_k the number of live processors is A_k and RANDOMSAMPLING is performed at the beginning of VS_k, then the probability of each sample hitting a live processor will be upper-bounded by $p_a = A_k/n$. Thus, $\mathbb{P}[X_{i,j} = 1] \leq p_a$, and $\mathbb{P}[X_{i,j} = 0] \geq 1 - p_a$. Since RANDOMSAMPLING is a relatively fast routine (with respect to the fact that the faults are considered to be uniformly distributed along the simulation process), p_a may be considered to be a very good approximation of the actual ratio of live processors in $\mathcal{RM}$ at the end of RANDOMSAMPLING. Thus, these indicator variables are clearly i.i.d. (random samples with replacement from the same sample space) with success probability very close to p_a, which implies that $\mathbb{E}[X_{i,j}] = p_a$ and $\mathrm{Var}[X_{i,j}] = p_a \cdot (1 - p_a)$, $\forall i \in [\mathtt{N}]$, $j \in [n]$. Hence, P_j's first estimation of the ratio of live processors in $\mathcal{RM}$ is given by the following relation: $R_j^{(0)} = 1/\mathtt{N} \cdot \sum_{i=1}^{\mathtt{N}} X_{i,j}$. The expected value and the variance of this new random variable are

$$\begin{aligned} \mathbb{E}[R_j^{(0)}] &= p_a \quad \text{and} \quad \mathrm{Var}[R_j^{(0)}] \\ &= \frac{1}{\mathtt{N}^2} \cdot \left(\sum_{i=1}^{\mathtt{N}} \mathrm{Var}[X_{i,j}] - 2 \cdot \sum_{i>k} \mathrm{Cov}(X_{i,j}, X_{k,j}) \right) = \frac{p_a \cdot (1 - p_a)}{\mathtt{N}}. \end{aligned}$$

Step 2. Each live processor P_j randomly selects processors from $\mathcal{RM}$ until it gets $\mathtt{N}$ live answers (this implies that each live processor will have to send $c/(1-a) \cdot \mathtt{N}$ polling messages to randomly chosen targets, and will accept the first $\mathtt{N}$ answers). The new estimation of P_j if $R_j^{(1)} = 1/\mathtt{N} \cdot \sum_{i=1}^{\mathtt{N}} R_{\lambda_i}^{(0)}$ (P_j has accepted the answers of $P_{\lambda_1}, P_{\lambda_2}, \ldots, P_{\lambda_{\mathtt{N}}}$). For this new estimation we have

$$\mathbb{E}[R_j^{(1)}] = p_a \quad \text{and} \quad \mathrm{Var}[R_j^{(1)}] = \frac{1}{\mathtt{N}^2} \cdot \left(\sum_{i=1}^{\mathtt{N}} \mathrm{Var}[R_{\lambda_i}^{(1)}] - 2 \cdot \sum_{i>k} \mathrm{Cov}(R_{\lambda_i}^{(0)}, R_{\lambda_k}^{(0)}) \right).$$

Since $R_{\lambda_1}^{(0)}, P_{\lambda_2}^{(0)}, \ldots, R_{\lambda_{\mathtt{N}}}^{(0)}$ are variables randomly chosen (with replacement) from the same sample space, they are independent from each other, and this implies that $\mathrm{Cov}(R_{\lambda_i}^{(0)}, R_{\lambda_k}^{(0)}) = 0$, $\forall j, k \in [\mathtt{N}]$. Hence, we have that

$$\mathrm{Var}[R_j^{(1)}] = \frac{1}{\mathtt{N}^2} \cdot \sum_{i=1}^{\mathtt{N}} \mathrm{Var}[R_{\lambda_i}^{(0)}] = \frac{p_a \cdot (1 - p_a)}{\mathtt{N}^2}. \tag{5}$$

Step 3. Repeat step 2 x times. After these x repetitions we have

$$\mathbb{E}[R_i^{(x+1)}] = p_a \quad \text{and} \quad \text{Var}[R_j^{(x+1)}] = \frac{p_a \cdot (1 - p_a)}{\mathsf{N}^{x+2}}.$$

Each processor P_j that is live estimates the number of live processors in $\mathcal{RM}$ at the beginning of the Virtual Superstep VS_k as $A_{j,k} = n \cdot R_j^{(x+1)}$, where

$$\mathbb{E}[A_{j,k}] = n \cdot p_a \quad \text{and} \quad \text{Var}[A_{j,k}] = n^2 \cdot \text{Var}[R_j^{(x+1)}] = \frac{n^2 p_a (1 - p_a)}{\mathsf{N}^{x+2}} \leq \frac{n^2}{4\mathsf{N}^{x+2}}.$$

In order to have small deviations among the live processors' estimations, the following must hold:

$$\lim_{n \to \infty} \frac{n^2}{4\mathsf{N}^{x+2}} = 0 \quad \Rightarrow \quad x = \mathcal{O}\left(\frac{\log n}{\log \mathsf{N}}\right), \tag{6}$$

considering that $\mathsf{N} = \log n$. For example, if $x = c \log n / \log \mathsf{N} - 2$, then $\text{Var}[A_{j,k}] = n^{-c+2}/4$, and by applying Chebysev's inequality we have $\mathbb{P}[|A_{j,k} - p_a \cdot n| \geq 1] \leq n^{-2c+4}/16$. □

Lemma 3.2. *The time cost of* BSPAGREEMENT *is* $T_{\text{BSPAgreement}} = \mathcal{O}(\log n / \log \log n \cdot \max\{L, g \cdot \log n\})$.

Proof. Each RANDOMSAMPLING demands the implementation of an $\mathcal{O}(\log n)$-relation (each live processor sends $\mathcal{O}(\log n)$ messages, while it may receive more than $\mathcal{O}(\log n)$ messages with polynomially small probability—an immediate application of Lemma 2.2) in a first superstep, and another $\mathcal{O}(\log n)$-relation (each polled live processor replies only to its own requesters) in a second superstep. Thus, two $\mathcal{O}(\log n)$-relations have to be implemented for RANDOMSAMPLING that cost time $\mathcal{O}(\max\{L, g \cdot \log n\})$. Additionally, the number of rounds that are necessary for the live processors to acquire consistent estimations of A_k, has been proved to be $\mathcal{O}(\log n / \log \log n)$ (considering that $\mathsf{N} = \log n$), while the cost of each round is dominated by the cost of RANDOMSAMPLING. □

3.5. *The Major Techniques Exploited by Robust-$\mathcal{BSP}$*

In this section we present the major techniques that are used by the Robust-$\mathcal{BSP}$ simulation strategy. These techniques have to do with the simulation process itself (Section 3.5.1), the periodic synchronization of the remaining live processors (Section 3.5.2), and the mixed storage scheme (Section 3.5.3).

3.5.1. *Primary and Secondary Jobs*

Primary job. Suppose that the current virtual superstep that has to be executed is VS_k. As mentioned above, the task of PRIMARYJOB (Figure 5) is to let all live processors behave as if everything were fine and no new processor failures will occur during VS_k.

[1] BSPAGREEMENT: Find an estimation of the currently live processors, A_k.
[2] **IF** $|PJ_i| > c \cdot n/A_k$
THEN Discard some of the assigned threads at random, until $|PJ_j| \leq c_1 \cdot /A_k$.
[3] **LC-phase:** Execute all the LC-phases of the threads in PJ_i.
[4] **Comm-phase:** Send the outgoing messages to their physical destinations.
[5] Register the undelivered messages to the List of Undelivered messages, LoU_i.
[6] SDOU: Deliver as many as possible of the messages in LoUs indirectly.
[7] Store the VLMs of all the completed threads using LOCALSTORAGE.

Fig. 5. The PRIMARYJOB procedure.

In this phase, each live processor P_i executes its own (unique) chunk of threads, held in PJ_i. Before starting the execution of the threads held in PJ_i, P_i upper-bounds the workload that it will execute (based on an estimation A_k of the currently live processors). Let $\mu_k \equiv c_1 n/A_k$ denote the maximum workload of each PrimaryJobQueue after the discarding operation. Consequently, each live processor executes its own portion of work, sends the outgoing messages to their physical destinations, and tries (employing the SDOU routine) to forward indirectly the messages heading for migrated threads. Finally, it stores the VLM of each completed thread using the LOCALSTORAGE scheme.

Lemma 3.3. *The cost of* PRIMARYJOB *that is executed at the beginning of the virtual superstep VS_k is given by*

$$T_{\text{PrimJob}} = \mathcal{O}(\mu_k \cdot \max\{h \cdot \log n, \log^4 n, T_{\text{LC}} + T_{\text{Comm}}\}). \tag{7}$$

Proof. For the time analysis of PRIMARYJOB we notice the following: step [1] is an application of the BSPAGREEMENT protocol, so that all the live processors acquire the same estimation of the number of live processors, A_k. The time cost for this step is $\mathcal{O}(\log n/\log\log n \cdot \max\{L, g \cdot \log n\})$, as shown in Section 3.4 presenting the BSPAGREEMENT protocol. Step [2] is trivial, while steps [3] and [4] are the same as if the input algorithm $\mathcal{A}$ ran directly on $\mathcal{VM}$, and impose no multiplicative overhead to the simulation process. Step [5] is also trivial, while step [6] costs $\mathcal{O}(h \cdot \mu_k \cdot \log n)$ (for this time estimation, see the time analysis of the SDOU procedure in Lemma 3.8). Finally, the LOCALSTORAGE procedure in step [7] costs at most $2\mu_k \cdot |\mathsf{VLM}| \cdot (\mathsf{N}_u + \max\{L, g\})$ according to the discussion about the safe storage of the CurrentStatus (see Section 3.5.3). Considering that $|\mathsf{VLM}| = \mathcal{O}(\log^2 n)$ and $\mathsf{N}_u \leq \log^2 n$ with high probability (see Section 3.5.2 for BACKTRACK occurrences), we get the stated cost for the PRIMARYJOB. □

Remark. Recall that N_u is a parameter that only depends on the total number u of BACKTRACK operations having occurred until the end of the current epoch.

Secondary job. As previously mentioned, the goal of each SECONDARYJOB J_i is to complete the outstanding work of VS_k and redistribute it among the live processors as evenly as possible (see Figure 6). The new work assignment is done as follows: Each of the A_k live processing elements chooses $\mathbf{b} = g(n) \cdot n/A_k$ threads to cover (that is,

[1] BSPAGREEMENT: Check for new LOCALSTORAGE corruptions.
[2] Each live processor P_i creates a bucket $\mathcal{B}_i$ of **b** randomly chosen threads.
[3] INFORMATIONGATHERING: For all the threads in $\mathcal{B}_i$, get the corresponding VLMs from the appropriate neighborhoods.
[4] **LC-phase:** Execute the LC-phases of the threads in $\mathcal{B}_i$.
[5] **Comm-phase:** Send all the outgoing messages to their physical destinations.
[6] Register the undelivered messages to LoU_i.
[7] SDOU: Send as many of the undelivered messages as possible, indirectly.
[8] Store the VLMs of the just completed threads using LOCALSTORAGE.

Fig. 6. The SECONDARYJOB procedure.

to execute, if not yet completed) either at random or derived by previous unsuccessful communication attempts with pending threads. For the purposes of the following analysis we consider all these choices independent and random, although it is our strong belief that the "biased" choices would make the processing elements focus exactly on the remaining unsatisfied threads.

Completeness of a secondary job. In this subsection we study the probability of the SECONDARYJOB J_i leaving pending threads (and thus not being able to complete the work of VS_k) when it finishes. For this we need the following technical lemma.

Lemma 3.4. *The number of individual processors that cover a specific* (*pending*) *thread during J_i is at least $c_3 \log n$ with probability $1 - n^{-c_4}$, $\forall c_3 > 0$, and c_4 depending on c_3.*

Proof. This lemma is an application of Lemma 2.2 with $c_3 = k_1$ and $m = g(n) \cdot n/A_k = n(k_1 \log n - 1)/A_k$. □

We now focus our attention on finding the probability of a specific thread being pending at the end of the ith SECONDARYJOB, J_i.

Lemma 3.5. $\forall c_5 > 0, \mathbb{P}[\mathcal{P}_i \geq c_5] \leq n^{-\min\{c_3 \cdot \log|(1-a)/r|, c_4\}+1}/c_5$.

Proof. For this fact to hold for thread T_m, all live processors that cover T_m must die during J_i. However, these processors are randomly chosen among A_k processors, at most $F_k \leq rn$ of which will die during the whole current virtual superstep. Hence, the probability of a specific random choice being a newly dead processor is at most $F_k/A_k \leq r/(1-a)$. As a consequence, the probability of T_m being pending at the end of J_i is bounded by the product of the probability of T_m having been covered by a specific number of (randomly chosen) processors, times the probability of all these processors dying during the current virtual superstep. By conditioning on the number of processors

that cover T_m, we have that

$$\mathbb{P}[T_m \text{is pending at the end of } J_i] \leq (1-n^{-c_4}) \cdot \left(\frac{r}{1-a}\right)^{c_3 \log n} + n^{-c_4} \cdot \mathbf{1}$$
$$\leq 2 \cdot n^{-\min\{c_3 \log(1-a)/r, c_4\}}.$$

Using n indicator variables

$$Z_m = \begin{cases} 1, & \text{if } T_m \text{ becomes pending during } J_i, \\ 0, & \text{otherwise,} \end{cases}$$

and setting $Z = \sum_{m=1}^{n} Z_m$, it is clear that the number of pending threads at the end of J_i is $\mathcal{P}_i = Z$ and $\mathbb{E}[\mathcal{P}_i] = \mathbb{E}[Z] = \sum_{m=1}^{n} \mathbb{E}[Z_m] \leq 2 \cdot n^{-\min\{c_3 \cdot \log((1-a)/r), c_4\}+1}$. By applying the Markov inequality on this expectation we get the desired result. □

As has been shown up to this point, the probability of having even just one pending thread at the end of a SECONDARYJOB is polynomially small.

Lemma 3.6. *The time cost of each* SECONDARYJOB *during the virtual superstep* VS_k *is given by*

$$T_{\text{SecJob}} = \mathcal{O}(\mathbf{b} \cdot \max\{h \cdot \log n, \log^4 n, T_{\text{LC}} + T_{\text{Comm}}\}). \tag{8}$$

Proof. Step [1] in SECONDARYJOB takes $\mathcal{O}(\log^2 n)$ according to the analysis of the BSPAGREEMENT protocol (see Section 3.4). Step [2] is a trivial task that is performed locally by each live processor. Step [3] costs $\mathbf{b} \cdot (\mathsf{N}_u \cdot |\mathsf{VLM}| + 2 \cdot |\mathsf{VLM}| \cdot \max\{L, g\})$ time steps for retrieving the necessary information for the $\mathbf{b}$ threads residing in the buckets of the live processors during the current SECONDARYJOB. Considering that $\mathsf{N}_u \leq \mathcal{O}(\log^2 n)$, and $|\mathsf{VLM}| = \mathcal{O}(\log^2 n)$, this is equal to $\mathcal{O}(\mathbf{b} \cdot (\log^4 n + \log n \cdot \max\{L, g\}))$. Steps [4] and [5] cost at most $\mathbf{b} \cdot T_{\text{LC}}$ and $\mathbf{b} \cdot T_{\text{Comm}}$, respectively, for each live processor. Step [6] is a trivial local operation, while step [7] costs $\mathcal{O}(h \cdot \mathbf{b} \cdot \log n)$, according to the analysis of the SDOU procedure (see Lemma 3.8). Finally, the LOCALSTORAGE procedure in step [8] costs at most $2 \cdot \mathbf{b} \cdot |\mathsf{VLM}| \cdot (\mathsf{N}_u + \max\{L, g\}) = \mathcal{O}(\mathbf{b} \cdot \log^4 n)$. Thus, the overall cost of each SECONDARYJOB is given by (8). □

Secure delivery of undelivered messages. In the SDOU procedure the primary objective is the safe delivery of messages heading for migrated threads that have been kept in the LoUs of the live processors up to this point. Additionally, in the case of a SECONDARYJOB having preceded, this is also the rescheduling procedure for the pending threads.

The basic idea of this procedure (see Figure 7) is the use of multiple **mailboxes** for each thread until all threads have been served by at least one live mailbox. That is, using ν random permutations of $[n]$, $(\Pi_1, \Pi_2, \ldots, \Pi_\nu)$, we proceed in ν rounds of indirect communication attempts, where in round j each live processor P_i tries to send any message for T_m, held in LoU_i, successfully to the mailbox $P_{\Pi_j(m)}$. Consequently (in the same round), each mailbox sends the incoming messages of the corresponding thread to a (possibly one chosen among many) requester of T_m according to the following

[1] **PARDO** ν times:
[2] **BSPRandomChoice:** Each live processor chooses a common random seed.
[3] Choose a new permutation Π_j according to the random seed and send all the messages to their corresponding mailboxes.
[4] **ThreadRequest:** Each processor claims the incoming messages for its own newly executed threads from the proper mailboxes (e.g., T_m from $P_{\Pi_j(m)}$).
[5] **ThreadAssignment:** Each live mailbox accepts ONLY one of the requests and sends the incoming messages of the thread that it serves to it.
[6] **ThreadCommittment:** Each live processor P_i that receives the mailbox of a thread T_m, adds it to PJ_i and marks it as newly completed.
[7] **ENDPARDO**.

Fig. 7. The SDoU procedure.

rescheduling rule: In the case of multiple contesting processors for a specific thread, the mailbox sends it to one of the contestants (and thus assigns the corresponding virtual processor to it) depending on the contestants' current workload. Let Π_0 be the identical permutation of $[n]$ and let $\Pi_1, \Pi_2, \ldots, \Pi_\nu$ be the required permutations so that each of the n threads is served by at least one live processor (to be its mailbox). For the following proof we need some definitions from the Theory of Negative Dependence of Random Variables (see [14] and [13]):

Definition 3.2. Let n be a positive integer.

1. The random variables $J_1, \ldots, J_n$ are said to have the **permutation distribution on** $[n]$ if they take values in $[n]$ and for any permutation $\sigma : [n] \to [n]$, $\mathbb{P}[J_1 = \sigma(1), \ldots, J_n = \sigma(n)] = 1/n!$.
2. Let $x_1, \ldots, x_n$ be arbitrary real numbers. The random variables $X_1, \ldots, X_n$ are said to have a **permutation distribution on** $\{x_1, \ldots, x_n\}$ if there is a set of random variables $J_1, \ldots, J_n$ with the permutation distribution on $[n]$ and $X_i = x_{J_i}, \forall i \in [n]$.

Definition 3.3. The random variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ are **negatively associated** if for every index set $I \subseteq [n]$, $\mathrm{Cov}[f(X_i, i \in I), g(X_j, j \in [n] - I)] \leq 0$, for all nondecreasing functions $f\colon \mathbb{R}^{|I|} \to \mathbb{R}$ and $g\colon \mathbb{R}^{|[n]-I|} \to \mathbb{R}$.

The following proposition concerning sets of negatively associated random variables is proved in [13] and will be used in what follows:

Proposition 3.1. *If* $\mathbf{X} = (X_1, \ldots, X_n)$ *and* $\mathbf{Y} = (Y_1, \ldots, Y_m)$ *are two sets of negatively associated random variables and are mutually independent, then the augmented vector* $(\mathbf{X}, \mathbf{Y}) = \mathbf{X} = (X_1, \ldots, X_n, Y_1, \ldots, Y_m)$ *is also a vector of negatively associated random variables.*

Lemma 3.7. *If the number ν of required rounds of communication attempts during* SDoU *is* $((c_6+1)/\log(1/a))\cdot\log n$*, then the failure probability is* $\mathbb{P}[\text{SDoU } \textit{fails}] \leq n^{-c_6}, \forall c_6 > 0$.

Proof. Consider the following $\nu \cdot n$ indicator variables:

$$\forall i \in [\nu], \quad j \in [n], \qquad X_{i,j} = \begin{cases} 1, & \text{if } P_{\Pi_i(j)} \text{ is live,} \\ 0, & \text{otherwise.} \end{cases}$$

For any fixed i, the vector $\mathbf{X}_i = (X_{i,1}, X_{i,2}, \ldots, X_{i,n})$ has the permutation distribution on

$$\textbf{Status} = \{\textit{Status}(1), \textit{Status}(2), \ldots, \textit{Status}(n)\},$$

where $\textit{Status}(j)$ indicates the status of machine j. Thus, for any fixed $i \in [\nu]$, the vector $\mathbf{X}_i$ follows the negative association property. Additionally, the vector $\mathbf{X} = (X_1, X_2, \ldots, X_\nu)$ consists of mutually independent components (since they correspond to randomly and independently chosen permutations), and thus $\mathbf{X}$ also has the negative association property.

Consider now the functions $Z_j = \sum_{i=1}^{\nu} X_{i,j}$. Since these variables are actually nondecreasing functions on disjoint sets of negatively associated random variables (r.v.'s in short), the vector $\mathbf{Z} = (Z_1, \ldots, Z_n)$ also follows the negative association property. Now we can proceed with the failure probability of the SDoU routine to serve all the assigned or pending threads successfully:

$$\mathbb{P}[\text{SDoU fails}] = \mathbb{P}[\exists j \in [n] : Z_j = 0] \leq \sum_{j=1}^{n} \mathbb{P}[Z_j = 0],$$

$$\mathbb{P}[Z_j = 0] = \mathbb{P}[X_{i,j} \leq 0, \forall i \in [\nu]] = \mathbb{P}[\bar{X}_{i,j} \geq 0, \forall i \in [\nu]].$$

Since $(X_{1,j}, \ldots, X_{\nu,j})$ are negatively associated, the same goes for the vector of complementary random variables $(\bar{X}_{1,j}, \ldots, \bar{X}_{\nu,j})$. Thus we have

$$\left.\begin{array}{l} \mathbb{P}[\bar{X}_{i,j} \geq 1, \forall i \in [\nu]] \leq \prod_{i\in[\nu]} \mathbb{P}[\bar{X}_{i,j} \geq 1] \\ \forall i \in [\nu], \quad j \in [n], \qquad \mathbb{P}[\bar{X}_{i,j} \geq 1] = \mathbb{P}[X_{i,j} \leq 0] \leq a \end{array}\right\} \quad \Rightarrow \quad \mathbb{P}[Z_j = 0] \leq a^{\nu}. \tag{9}$$

The total failure probability of SDoU is bounded by $\mathbb{P}[\text{SDoU fails}] \leq n \cdot a^{\nu} = n^{-c_6}$, where $n \cdot a^{\nu} = n^{-c_6} \Rightarrow \nu = ((c_6+1)/\log(1/a)) \cdot \log n$ □

Lemma 3.8. *The time cost of* SDoU *is* $T_{\text{SDoU}} = \mathcal{O}(\log n \cdot \max\{l, g\mathbf{b}h, g\mu_k h\})$.

Proof. For the time analysis of SDoU we have: Step [1] implies $\mathcal{O}(\log n)$ rounds, according to Lemma 3.7. For Step [2] we consider that there is a BSP **Random Number Generator** that provides at start-up all live processors of $\mathcal{RM}$ with a string of random seeds. Then each processor can fix the new permutation of step [3] using the next seed of this "shared" string of random seeds. Steps [4] and [5] are actually implementations

of $(h \cdot \max\{\mathbf{b}, \mu_k\})$-relations for the requests of the inboxes of the threads covered by a specific processor, and the assignments of the pending threads to one of the contestants. Step [6] is a trivial local operation. □

The failure probability of a virtual superstep. In this section we study the failure probability of a single virtual superstep, VS_k, and specify the number of K_0 of the virtual supersteps that comprise an epoch.

Lemma 3.9. *The failure probability of a Virtual Superstep VS_k to complete its work is $n^{\mathcal{O}/-\log\log n)}$, considering that $y = \log\log n$* SECONDARYJOBS *are executed, if necessary.*

Proof. A virtual superstep VS_k fails if all the y SECONDARYJOBS that it performs have at least one pending thread when they finish, or the corresponding executions of SDOU fail to serve an assigned thread. In this section we neglect the corruption probability of some neighborhood of real processors (which implies a failure of some LOCALSTORAGES) because this failure probability is separately studied for the whole epoch in Section 3.5.2 that deals with BACKTRACK occurrences. Thus, we consider the INFORMATIONGATHERING and LOCALSTORAGE routines to be safe. In that case, VS_k will fail if, for all its SECONDARYJOBS, there is either a failure to cover all the threads or a failure for some (pending or assigned) threads to be served by a live mailbox during the SDOU procedure. The probability of SDOU leaving an assigned or pending thread with no operational mailbox has already been estimated and is $\mathbb{P}[\text{SDOU fails}] \leq n^{-c_6}$, while the probability that a specific SECONDARYJOB fails to cover at least an assigned or pending thread is $\mathbb{P}[SJ_i \text{ fails}] \leq 2 \cdot n^{-\min[c_3 \cdot \log((1-a)/r), c_4+1}$. Thus, the failure probability for each SECONDARYJOB to finish the work of VS_k is polynomially small and the total failure probability for VS_k is

$$\mathbb{P}[VS_k \text{ fails}] = n^{-\mathcal{O}(c \cdot y)}, \tag{10}$$

where y is the number of SECONDARYJOBS to be executed and $c = \min\{c_6, c_3 \cdot \log((1 - a)/r) - 1, c_4 - 1\}$. □

Remark. Having a subpolynomially small failure probability for each of the virtual supersteps that comprise an epoch, it is now apparent that an epoch can contain $K_0 = \Theta(n)$ virtual supersteps and still have subpolynomially small failure probability.

3.5.2. *Checkpointing and Backtracking.* The CHECKPOINT procedure is actually a virtual process that is done by any processor during the simulation of the input algorithm. More specifically, CHECKPOINT signifies the failure of some LOCALSTORAGE or INFORMATIONGATHERING procedure call discovered by a live processor, either because of an update failure of the LOCALSTORAGE routine or because of an unsuccessful attempt to retrive a specific VLM. The discovery of a problematic situation is disseminated to the rest of the live processors the next time that a BSPAGREEMENT protocol is executed as an exception code to the specific live processor's value. Observe that a new processor failure may not cause any trouble at all because the problematic situation will be discovered by

$\mathcal{O}(\log n)$ live processors that will try to cover the corresponding pending thread (due to the new death) during the next SECONDARYJOB.

When such an interruption to the flow of the simulation process is done, the BACKTRACK operation simply makes the live processors set their program counters to the last SafeState, set a new value $\mathtt{N}_u = 2 \cdot \mathtt{N}_{u-1}$ (u is the number of BACKTRACKS having occurred up to now), consider a new, random hash function for the size-$\mathtt{N}_u$ equipartition of the real processing elements into $n/\mathtt{N}_u$ neighborhoods of real processors (see the description of the LOCALSTORAGE scheme in Section 3.5.3) and continue with the simulation process after having retrieved the CurrentStatus from the last SafeState.

For an epoch e_i to be successfully simulated, K_0 consecutive virtual supersteps must be executed with no BACKTRACK interference. The epoch will end up with the creation of a new SafeState. In what follows we study the corruption probability of some neighborhood of processors because of the fault occurrences up to now and the number of BACKTRACKS that may occur during an epoch of K_0 virtual supersteps.

Remark. The choice of a new (pseudo)random hash function for the construction of the new neighborhoods of processors is done in order to redistribute the faults occurred up to this point evenly among the new neighborhoods, and protect the simulation from a malicious behavior of an adversary that would try to focus his power on a single neighborhood of processors.

Backtrack occurrences during an epoch. In this subsection we bound the number of corruptions that may occur in a single epoch of virtual supersteps. First we estimate the corruption probability at a specific instance of the simulation process, and consequently we calculate the total number of corruptions.

Lemma 3.10. *The probability* $\Phi(D, \mathtt{N})$ *that a randomly chosen size-*$\mathtt{N}$ *equipartition corrupts because of* D *processor failures is given by*

$$\Phi(D,\mathtt{N}) = \begin{cases} 0, & \text{if} \quad D < \dfrac{\mathtt{N}}{2} + 1, \\ \dfrac{n \cdot 2^{\mathtt{N}+1/2}}{\sqrt{\pi \mathtt{N}} \cdot (\mathtt{N}+2)} \cdot \exp\Big[-\Big(\dfrac{\mathtt{N}}{2}+1\Big)(H_n - H_D)\Big], & \text{otherwise}, \end{cases}$$

where H_n *and* H_D *are the corresponding harmonic numbers.*

Proof. Each group of the equipartition to be constructed may be thought of as a bin that will receive exactly $\mathtt{N}$ balls, some of which are expected to be black (dead processors) and the remainder will be red (live ones). The amount of black balls is D, and the red ones is $A = n - D$. All possible ways of creating a size-$\mathtt{N}$ equipartition of n indistinguishable balls are $n!/(\mathtt{N}!)^{n/\mathtt{N}}$. We count all possibly corrupted equipartitions by the D black balls, in a constructive fashion. More specifically, we first choose one of the bins at random. Then we choose $\mathtt{N}/2 + 1$ from the D black balls and throw them in the chosen bin. Consequently, we choose $\mathtt{N}/2 - 1$ from the remaining $n - \mathtt{N}/2 - 1$ balls and throw them

into the chosen bin. Finally, we let the remaining $n - \mathtt{N}$ balls be equipartitioned into the $n/\mathtt{N} - 1$ bins in all possible ways. In numbers, all possible corrupted equipartitions are

$$\frac{n}{\mathtt{N}} \cdot \binom{D}{\mathtt{N}/2+1} \cdot \binom{n-\mathtt{N}/2-1}{\mathtt{N}/2-1} \cdot \frac{(n-\mathtt{N})!}{(\mathtt{N}!)^{n/\mathtt{N}-1}}.$$

Hence, the requested corruption probability (provided that $D \geq \mathtt{N}/2+1$) is given by

$$\begin{aligned}
\Phi(D,\mathtt{N}) &= \frac{n}{\mathtt{N}} \cdot \binom{D}{\mathtt{N}/2+1} \cdot \binom{n-\mathtt{N}/2-1}{\mathtt{N}/2-1} \cdot \frac{(n-\mathtt{N})!}{(\mathtt{N}!)^{n/\mathtt{N}-1}} \cdot \frac{(\mathtt{N}!)^{n/\mathtt{N}}}{n!} \\
&= \frac{n}{\mathtt{N}} \cdot \binom{D}{\mathtt{N}/2+1} \cdot \binom{n-\mathtt{N}/2-1}{\mathtt{N}/2-1} \cdot \binom{n}{\mathtt{N}}^{-1} \\
&= \frac{n \cdot D! \cdot (n-\mathtt{N}/2-1)! \cdot \mathtt{N}! \cdot (n-\mathtt{N})!}{\mathtt{N} \cdot (\mathtt{N}/2+1)! \cdot (D-\mathtt{N}/2-1)! \cdot (\mathtt{N}/2-1)! \cdot (n-\mathtt{N})! \cdot n!} \\
&= \cdots = \frac{n \cdot \mathtt{N}!}{(\mathtt{N}+2) \cdot (\mathtt{N}/2)!^2} \cdot \prod_{\lambda=D+1}^{n} \left(1 - \frac{\mathtt{N}+2}{2\lambda}\right) \\
&\sim \frac{n \cdot \mathtt{N}!}{(\mathtt{N}+2) \cdot (\mathtt{N}/2)!^2} \cdot \exp\left[-\left(\frac{\mathtt{N}}{2}+1\right) \cdot (H_n - H_D)\right]. \qquad (11)
\end{aligned}$$

By applying Stirling's formula, one can show that the claimed result holds. In the case that $D \leq \mathtt{N}/2+1$, there is no chance of having a corrupted neighborhood of real processors. □

The following lemma states that it is most unlikely to have more than $\log\log n$ BACKTRACK occurrences during the whole simulation process (this also implies that the size of the neighborhoods will be at most $\mathcal{O}(\log^2 n)$ with subpolynomially small failure probability):

Lemma 3.11. *The probability of a neighborhood corruption after* $\log\log n$ BACKTRACK *is* $\mathcal{O}(n^{-\log n} \cdot \log^{-3} n)$.

Proof. Assume that we have n processors and we choose a random equipartition of them into $n/\mathtt{N}_u$ neighborhoods of size $\mathtt{N}_u$ (u indicates the number of BACKTRACK occurrences, up to now). Suppose also that D processors have already died (and have caused the u BACKTRACKS) and $A = n - D$ remain live. We say that an equipartition **corrupts** if there exists a neighborhood that has at least $\mathtt{N}_u/2 + 1$ dead processors.

The corruption probability of size-$\mathtt{N}_u$ equipartition ($\mathtt{N}_u = 2^u \cdot \log n$) is given by $\Phi(D, N_u)$, which is always at most equal to $\Phi(an, N_u)$. Some calculation will help to see that this is a very good bound for the failure probability, which actually implies that at most $\log\log n$ backtrack operations may occur during the simulation of the input algorithm, with very high probability. More specifically, considering that $H_n - H_{a \cdot n} =$

$\ln(1/a) + \Theta(1) = \ln(1/a) + \omega_1$, we have

$$\begin{aligned}\Phi(D, \mathsf{N}_u) \le \Phi(an, \mathsf{N}_u) &\le \frac{n \cdot 2^{\mathsf{N}_u+1/2}}{\sqrt{\pi \mathsf{N}_u} \cdot (\mathsf{N}_u + 2)} \cdot \exp\left[-\left(\frac{\mathsf{N}_u}{2} + 1\right)\left(\ln\left(\frac{1}{a}\right) + \omega_1\right)\right] \\ &\le \frac{1}{\sqrt{\pi \mathsf{N}_u} \cdot (\mathsf{N}_u + 2)} \cdot 2^{\mathsf{N}_u + 1/2 + \log n - \mathsf{N}_u \cdot (\log(1/a)/2 + \omega_1/(2\ln 2)) - (\log(1/a) + \omega_1/\ln 2)} \\ &\le \frac{2^{\omega_2}}{\sqrt{\pi \mathsf{N}_u} \cdot (\mathsf{N}_u + 2)} = \mathcal{O}\left(n^{-2^u} \cdot (2^u \log n)^{-3/2}\right), \qquad (12)\end{aligned}$$

where

$$\begin{aligned}\omega_2 \equiv \mathsf{N}_u + \tfrac{1}{2} + \log n - \mathsf{N}_u \cdot \left(\frac{\log(1/a)}{2} + \frac{\omega_1}{2\ln 2}\right) \\ - \left(\log\left(\frac{1}{a}\right) + \frac{\omega_1}{\ln 2}\right) = \mathcal{O}(-2^u \cdot \log n).\end{aligned}$$

□

3.5.3. *Safe Storage of $\mathcal{VM}$'s* CurrentStatus. In this subsection we study the robustness of our simulation strategy against the (at most $a \cdot n$) processor failures that are imposed by the parallel setting. Recall that our parallel setting is a BSP machine which is essentially a message passing model. Thus, for overcoming arbitrary processor failures, one has to sustain some information replication which depends on the input parameter a (recall that a is an upper bound on the fraction of processor failures that may occur overall during the simulation process). Since the recovery from a situation where $a \cdot n$ faults have occurred necessitates the existence of an (at least) $(1/(1-a))$-fold replication of the VLMs that comprise the CurrentStatus of $\mathcal{VM}$, one might consider that the best that can be done is to use the IDA algorithm which achieves space optimality, given the total fraction of faults, a. Yet, this is a very expensive procedure to run in each virtual superstep, since IDA would require at least $\mathcal{O}(n^2)$ time (according to the time estimations of the BSP implementation of IDA in Section 3.5.3) to create a SafeState from which up to $a \cdot n$ faults may be overcome. Additionally, such a storage scheme would force a total cost of the simulation process that would not be scalable with the number of fault occurrences and thus our strategy would be efficient if as many as possible errors actually occurred during the execution of BSP algorithm.

The salient point of our storage scheme (see Figure 8) is the use of a local, volatile storage routine (LOCALSTORAGE for the tentative storage of $\mathcal{VM}$'s CurrentStatus in combination with a global routine (GLOBALSTORAGE) that robustly stores periodic instances of the CurrentStatus at the end of each epoch. This way, the cost of this expensive operation of creating a new SafeState is amortized over the K_0 virtual supersteps that comprise a single epoch. The more volatile LOCALSTORAGE scheme during each virtual superstep makes the execution much faster, while being able to tolerate small bursts of processor failures in each neighborhood of real processors. Each time a neighborhood corruption occurs, the size of the neighborhoods participating in LOCALSTORAGE is doubled and a new hash function for the determination of the neighborhoods is employed. Additionally, an INFORMATIONGATHERING procedure takes over the responsibility of retrieving the requested information, either from the proper neighborhoods of real processors or from the last SafeState in the case when a BACKTRACK operation has preceded.

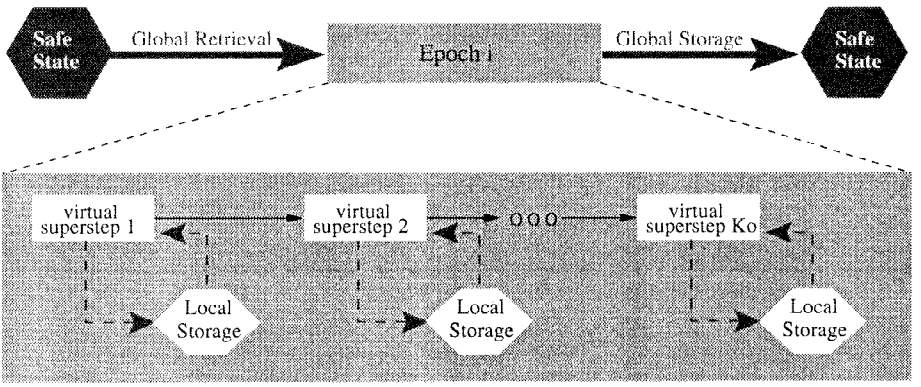

Fig. 8. The mixed storage scheme.

Local storage. The LOCALSTORAGE routine is used to save the VLM of each newly completed thread T_m (at the end of the current (either Primary or Secondary) Job) to a proper (with respect to the $\mathcal{V}id$ of T_m) size-N_u neighborhood, so that any $\mathsf{N}_u/2$ of them will be able to reconstruct in the future.

In particular, we consider that the n real processors are partitioned into n/N_u neighborhoods of size N_u according to a randomly chosen hash function $\mathcal{H}$: $[n] \to [n/\mathsf{N}_u]$. Then each neighborhood takes over the responsibility of storing the VLMs of the virtual processors that have their physical destinations in this neighborhood. So, if a live processor P_i uses LOCALSTORAGE to save a newly completed thread's VLM (e.g., VLM_j), then it will apply the FILEDISPERSAL part if IDA on VLM_j with $m = \mathsf{N}_u/2$ (see Section 3.5.3), and will save the N_u created files to the neighborhood of processors indicated by $\mathcal{H}(j)$.

Supposing that a real processor P_j has NC_i newly completed threads during job J_i, and considering the fact that $\log n \leq \mathsf{N}_u \leq \mathcal{O}(\log^2 n)$ with high possibility (recall the justification of this fact in Section 3.5.2), the time overhead on P_j for the LOCALSTORAGE will be

$$T_{\text{LocalStorage}} = 2 \cdot |\mathsf{VLM}| \cdot NC_i \cdot (\mathsf{N} + \max\{L, g\}). \tag{13}$$

The space overhead for each real processor P_j being a member of a specific neighborhood, will be the sum of the sizes of the dispersal files for the VLMs of the corresponding N_u virtual processors which are locally stored in the specific neighborhood, that is, $\mathsf{N}_u \cdot |\mathcal{F}_i| = \mathsf{N}_u \cdot |\mathsf{VLM}|/(\mathsf{N}_u/2) = 2 \cdot |\mathsf{VLM}|$, where $|\mathsf{VLM}|$ is the size of each of the virtual local memories. This implies a two-fold space overhead on each live processor for implementation of the LOCALSTORAGE scheme on each processor, which is optimal for tolerating up to $\mathsf{N}_u/2$ processor failures in each neighborhood.

Global storage. The purpose of the GLOBALSTORAGE scheme (Figure 9) is to create SafeState periodically, that is, to store periodic instances of $\mathcal{VM}$'s CurrentStatus securely for the simulation process to backtrack to, in case some (locally) unrecoverable errors have occurred. This routine is actually the one that necessitates the existence of the input

[1] **FOR** $k = 1$ **TO** Λ **DO**
[1.1] **IF** $P_{k \cdot \mathsf{N}_u + j}$ is live,
[1.2] **THEN** P_j sends to it its own part of the LocalStorage
[1.3] **ELSE** P_j makes at most $\mathsf{N}_u/2 - 1$ trials to find a live processor in the target neighborhood, to send its part of the LOCALSTORAGE.
[2] **PartDistribution:** Each live processor that holds multiple parts of a LOCALSTORAGE, distributes them to live processors of the same neighborhood that have no part of this LOCALSTORAGE.

Fig. 9. The GLOBALSTORAGE scheme.

parameter a that expresses an upper bound on the overall fraction of processor failures during the simulation of the input algorithm $\mathcal{A}$.

Supposing that we used IDA for the secure storage of the CurrentStatus of $\mathcal{VM}$ to the whole $\mathcal{RM}$, we would need an $\mathcal{O}(n^2)$ time overhead, according to the time estimations of the implementation of IDA on BSP, which is actually a prohibitive cost in a setting of parallel computations simulation. There are two reasons for this prohibitive cost in this approach for the implementation of GLOBALSTORAGE: the first is IDA's restriction for $|\mathsf{VLM}| \geq m = (1-a)n$ which necessitates the extension of the VLMs with some dummy characters, and the second is that we disperse once more the already dispersed (for the sake of the last LOCALSTORAGE) VLMs. Thus we resort to an alternative scheme for the GLOBALSTORAGE that tries to avoid the unnecessary time (and space, which is actually communication cost in BSP) overhead. The GLOBALSTORAGE strategy is as follows: Each of the n/N_u neighborhoods in $\mathcal{RM}$ creates replicas of the contents of the next Λ (modulo-n/N_u) neighborhoods. Thus, each real processor actually participates in $\Lambda + 1$ replicas of the LOCALSTORAGE scheme, and each size-N_u neighborhood is capable of restoring the VLMs of $(\Lambda + 1) \cdot \mathsf{N}_u$ distinct virtual processors. The space overhead for each real processor, regarding the GLOBALSTORAGE, is $(\Lambda + 1) \cdot \mathsf{N}_u \cdot |\mathsf{VLM}|$. As for the time overhead for creating the Λ new replicas of the last LOCALSTORAGE before the new GLOBALSTORAGE, this may be done in Λ communication supersteps, where each neighborhood sends the contents of its LOCALSTORAGE to a new neighborhood, and afterwards an "all-to-all" communication in each neighborhood re-assigns the parts of a specific LOCALSTORAGE to unique live processors.

Lemma 3.12. *Given that no more that $a \cdot n$ faults may occur in overall during the simulation of an input algorithm $\mathcal{A}$, the necessary number of replicas of the* LOCALSTORAGE *that will comprise a new* SafeState *is no more than* $2an/(\mathsf{N}_u + 2)$.

Proof. For a specific neighborhood of processors to be globally destroyed, all the $\Lambda + 1$ replicas should be destroyed by the fault occurrences. This means that at least $\mathsf{N}/2 + 1$ real processors per size-N neighborhood must have already died. Thus, totally, we must have at least $(\Lambda + 1) \cdot (\mathsf{N}/2 + 1)$ faults up to now. However, if

$$(\Lambda + 1) \cdot \left(\frac{\mathsf{N}}{2} + 1\right) > a \cdot n \quad \Rightarrow \quad \Lambda > \frac{2an}{\mathsf{N} + 2} - 1, \tag{14}$$

then it is impossible for this to happen. □

Notice that in each round the communication among the neighborhoods is "one-to-one." So, if we fix $\Lambda = 2an/(\mathsf{N}_u + 2)$, the time for creating a new SafeState with GLOBALSTORAGE is given by

$$T_{\text{GlobalStorage}} \leq 2 \cdot \Lambda \cdot |\mathsf{VLM}| \cdot \left(\frac{\mathsf{N}_u}{2} - 1 \right) \leq \mathcal{O}(a \cdot n \cdot |\mathsf{VLM}|). \tag{15}$$

GLOBALSTORAGE is called every K_0 virtual supersteps, because it is actually a time-consuming operation which introduces a heavy cost. This way this heavy cost is amortized among the K_0 virtual supersteps.

Information gathering. The INFORMATIONGATHERING procedure is an application of the FILERETRIEVAL phase of the IDA algorithm from the proper neighborhood of real processors (with respect to the requested threads' $\mathcal{V}id$s), unless a BACKTRACK has just preceded, in which case each VLM is retrieved from the last SafeState of $\mathcal{VM}$, which is robustly stored in $\mathcal{RM}$.

According to the analysis of the FILERETRIEVAL phase of IDA, each LOCALRETRIEVAL of a VLM costs

$$T_{\text{LocalRetrieval}} = \mathsf{N}_u \cdot |\mathsf{VLM}| + 2 \cdot |\mathsf{VLM}| \cdot \max\{L, g\}, \tag{16}$$

while a GLOBALRETRIEVAL operation will actually have to try to retrieve the specific VLM from one of the $\Lambda + 1$ replicas of the corresponding neighborhood of the LOCALSTORAGE scheme in $\mathcal{RM}$:

$$\begin{aligned} T_{\text{GlobalRetrieval}} &\leq (\Lambda + 1)(\mathsf{N}_u \cdot |\mathsf{VLM}| + 2|\mathsf{VLM}| \cdot \max\{L, g\}) \\ &= \mathcal{O}(a \cdot n \cdot |\mathsf{VLM}|). \end{aligned} \tag{17}$$

The following lemma summarizes the time costs for these routines that comprise our mixed storage scheme:

Lemma 3.13. *Supposing that each live processor is responsible for at most λ threads, the time costs for the procedures of our storage scheme are*

$$T_{\text{LocalStorage}} = 2\lambda|\mathsf{VLM}|(\mathsf{N} + \max\{L, g\}), \tag{18}$$

$$T_{\text{GlobalStorage}} = \mathcal{O}(\lambda an|\mathsf{VLM}|), \tag{19}$$

$$T_{\text{LocalRetrieval}} = \lambda(\mathsf{N}_u|\mathsf{VLM}| + 2|\mathsf{VLM}| \cdot \max\{L, g\}), \tag{20}$$

$$T_{\text{GlobalRetrieval}} = \mathcal{O}(\lambda an|\mathsf{VLM}|). \tag{21}$$

A BSP *implementation of IDA.* In this section we demonstrate the BSP implementation of the IDA algorithm (see Figure 10), as well as our routines for storing and retrieving the CurrentStatus of $\mathcal{VM}$ from the live processors of $\mathcal{RM}$.

Suppose that we have a file $\mathcal{F}$ of size $|\mathcal{F}|$ and we want to store it safely among N processing elements, in such a way that any m of these processors will be able to reconstruct $\mathcal{F}$. IDA is a space-optimal strategy for dispersing and reconstructing the initial file, which may very easily be adapted to the BSP model, since it was designed for applications to arbitrary distributed environments.

[1] **FileDispersal:** Processor P_i disperses $\mathcal{F}$ and sends the corresponding parts to the processing elements of the proper size-N neighborhood.
[1.1] P_i splits $\mathcal{F}$ into N parts $(\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_{\mathtt{N}})$ of size $|\mathcal{F}_j| = |\mathcal{F}|/m$, $j \in [\tilde{\mathtt{N}}]$ each.
[1.2] P_i sends the N parts to the processors of the size-N neighborhood.
[2] **FileRetrieval:** A processor P_i wants to gather the parts of a file $\mathcal{F}$ from a proper (size-N) neighborhood, and reconstruct $\mathcal{F}$.
[2.1] P_i sends requests to all the parts of the proper neighborhood.
[2.2] Every live processor in the neighborhood sends its own part to P_i.
[2.3] **IF** P_i has enough (i.e., more than m) parts, **THEN** it reconstructs $\mathcal{F}$.

Fig. 10. A BSP implementation of IDA.

For the computational cost of the file dispersal into N parts, IDA uses a set of N size-m vectors, that are m-wise independent, splits $\mathcal{F}$ into size-m sequences of characters, and applies an inner product operation on $\mathcal{F}$ with each of these N vectors, so as to produce the N new files, any m of which be able to reconstruct $\mathcal{F}$. An obvious restriction of IDA is that $|\mathcal{F}| \geq m$. If not, $\mathcal{F}$ is extended with some dummy characters, up to size of m words. The communication cost for the transmission of the N parts of $\mathcal{F}$ to the proper neighborhood is actually the cost of a "$1 \rightarrow \mathtt{N}$" transmission of size-$|\mathcal{F}_j|$ messages, that is the same as an $\mathtt{N} \cdot |\mathcal{F}_j|$-relation.

As for the file retrieval, IDA uses the inverse transformation to convert any m input files of size $|\mathcal{F}_i| = |\mathcal{F}|/m$ each, to the file $\mathcal{F}$. This is easily shown to cost $2m \cdot |\mathcal{F}|$ local operations, while the communication cost of this part is an N-relation for the requests, plus an "$\mathtt{N} \rightarrow 1$" transmission of the input files to P_i. This is the same as an $(\mathtt{N} \cdot |\mathcal{F}_j|)$-relation. According to the above discussion, the costs of IDA for Dispersal and Retrieval of the file $\mathcal{F}$ on a size-N neighborhood of a BSP machine are the following:

$$T_{\text{Dispersal-LC}} = \mathtt{N} \cdot \lceil |\mathcal{F}|/m \rceil \cdot (2m - 1), \tag{22}$$

$$T_{\text{Dispersal-Comm}} = \lceil |\mathcal{F}|/m \rceil \cdot \mathtt{N} \cdot \max\{L, g\}, \tag{23}$$

$$T_{\text{Retrieval-LC}} = 2m \cdot |\mathcal{F}|, \tag{24}$$

$$T_{\text{Retrieval-Comm}} = \lceil |\mathcal{F}|/m \rceil \cdot \mathtt{N} \cdot \max\{L, g\}. \tag{25}$$

Remark. The *ceiling* operation expresses the fact that $|\mathcal{F}| \geq m$. In fact this implies that the transmission of each of the new files to the N processors costs at least as much as the transmission of N constant-size messages.

3.6. *The Performance of the Robust-$\mathcal{BSP}$ Simulation Strategy*

In this section we estimate the amortized cost of a virtual superstep executed by our Robust-BSP simulation strategy, and give a bound on the competitive ratio of our strategy, against an optimal off-line strategy, that always lets the operational processors execute a fully balanced workload. Recall that the optimal workload would be n/A_k threads per operational processor, and thus $T_{\text{OPT}} = n/A_k \cdot (T_{\text{LC}} + T_{\text{Comm}})$.

Theorem 3.1. *The amortized cost for the simulation of a single virtual superstep is given by*

$$T_{VS_k} = \mathcal{O}\left((\log n \cdot \log\log n)^2 \cdot T_{\text{OPT}} + \text{polylog}(n)\right) \tag{26}$$

with probability at least $\mathcal{O}(1 - n^{-\log n} \cdot \log^{-3} n)$.

Proof. Assume that during the current epoch $\mathcal{B}$ corruptions of the LOCALSTORAGE occur. Then Robust-$\mathcal{BSP}$ will have to execute each of the K_0 supersteps of this epoch at most $\mathcal{B}+1$ times, and it will have to retrieve information from the last SafeState $\mathcal{B}$ times. So, the time of a single epoch of K_0 virtual supersteps is upper-bounded by the following equation:

$$\begin{aligned} T_{\text{epoch}} &= (\mathcal{B}+1)K_0(T_{\text{PrimJob}} + \log\log n \cdot T_{\text{SecJob}}) \\ &\quad + \mathcal{B} \cdot (\mu_k + \mathbf{b} \cdot \log\log n) \cdot (T_{\text{GlobalStorage}} + T_{\text{GlobalRetrieval}}), \end{aligned}$$

since each live processor will retrieve at most $\mu_k + \mathbf{b} \cdot \log\log n$ threads to execute from the last SafeState, each time the current epoch restarts. However, according to Lemma 3.11, the number of BACKTRACKS in the whole simulation process is no more than $\log\log n$, with probability at least $\mathcal{O}(1 - n^{-\log n} \cdot \log^{-3} n)$.

Notice also that if we suppose that $T_{\text{LC}} + T_{\text{Comm}} = \mathcal{O}(g \cdot h) \geq \log^4 n$, then $T_{\text{PrimJob}} = \mu_k \cdot \max\{h \cdot \log n, (T_{\text{LC}} + T_{\text{Comm}})\} = \mathcal{O}((\log n)/g \cdot T_{\text{OPT}})$ and $T_{\text{SecJob}} = \mathbf{b} \cdot \max\{h \cdot \log n, (T_{\text{LC}} + T_{\text{Comm}})\} = \mathcal{O}((\log^2 n)/g \cdot T_{\text{OPT}})$. In that case, the amortized cost of a single virtual superstep is given by

$$\begin{aligned} T_{VS_k} &= (\mathcal{B}+1) \cdot T_{\text{PrimJob}} + \frac{an \cdot \log\log n}{K_0} \cdot T_{\text{SecJob}} \\ &\quad + \frac{\mathcal{B} \cdot (\mu_k + \mathbf{b} \cdot \log\log n)}{K_0} \cdot (T_{\text{GlobalStorage}} + T_{\text{GlobalRetrieval}}) \\ &= \mathcal{O}((\log n \cdot \log\log n)^2) \cdot T_{\text{OPT}} + \mathcal{O}(\log^3 n \cdot (\log\log n)^2), \end{aligned}$$

where the additive polylog(n)-term is due to the fact that $K_0 = \Theta(n)$, $\mathcal{B} = \log\log n$, μ_k and $\mathbf{b} = \mathcal{O}(\log n)$, and $T_{\text{GlobalStorage}} + T_{\text{GlobalRetrieval}} = \mathcal{O}(n \cdot \log^2 n)$ (see Lemma 3.13). □

Remark. Since the probability of having more than $\log\log n$ BACKTRACK occurrences is $\mathcal{O}(n^{-\log n} \cdot \log^{-3} n)$, the expected (amortized) cost of each virtual superstep converges to the above value, since the subpolynomially small failure probability dominates over the cost of some extra BACKTRACK occurrences.

4. The Adaptive Load-Balancing Strategy

A major result of this work which is also of independent interest, is the proposes strategy for balancing the work of the n virtual processors among the currently live processors of $\mathcal{RM}$. In fact, this is an adaptive on-line load-balancing technique, since the sequence of

fault occurrences is not known a priori to our simulation process and the live processors should not execute more than a factor times the optimal workload in a stable state.

Starting from a balanced situation (at the last SafeState), we show in this section how we keep the work of the live processors balanced (in a tentative fashion) for a whole epoch of K_0 virtual supersteps. Recall that K_0 can be as large as $\Theta(n)$. Let $\mathcal{Z}_0(i,k)$ denote the size of P_i's queue after discarding the excessive work (in case of overloaded real processors), and let $\mathcal{Z}(i,k)$ denote its size at the end of VS_k. Our ADAPTIVELOAD-BALANCING scheme (ALB in short), which is inherent in the Robust-BSP simulation strategy, is the following:

(1) At the beginning of each virtual superstep VS_k, each overloaded real processor P_i cuts off the excessive work (i.e., keeps at most $c_1 n/A_k$ threads at random in PJ_i, according to the estimation A_k of the number of live processors in $\mathcal{RM}$, and makes the remaining threads pending). This is the Discarding step at the beginning of each PRIMARYJOB.

(2) All the pending threads of $\mathcal{VM}$ (due to either new deaths or discardings) are rescheduled to still live processors of $\mathcal{RM}$ as follows:

(α) Each live processor contests for $\mathbf{b} \cdot \log\log n$ randomly chosen threads to cover, during the $\log\log n$ SECONDARYJOB.

(β) Each mailbox that takes over a thread T_λ during a round of SDOU, assigns it (if it is pending) to one of the live processors that contests for it, with probability

$$\mathbb{P}[P_{\lambda_i} \text{ gets } T_\lambda] = \frac{1/\mathcal{Z}_0(\lambda_i,k)}{\sum_{j\in C}(1/\mathcal{Z}_0(j,k))},$$

where $C = \{P_{\lambda_1}, P_{\lambda_2}, \ldots\}$ is the set of processors contesting for T_λ.

Theorem 4.1. *Let $\xi > 1$ be a constant. Then the load of each live processor P_i in $\mathcal{RM}$ at the end of VS_k is $\mathcal{Z}(i,k) \le c_1 n/A_k \cdot \Theta(\log\log n)$, with probability at least $(1-n^{-\xi})^{(k+1)} \ge 1-(k+1)n^{-\xi}$.*

Proof. Let $\xi > 0$ be a constant. We shall prove the good behavior of ALB using induction on the epochs of the input algorithm $\mathcal{A}$.

Initial Step. At the beginning of our simulation process, each virtual processor is assigned to its physical destination, and thus, each operational processor in $\mathcal{RM}$ has exactly one thread in its PrimaryJobQueue.

Inductive Hypothesis. Suppose that at the beginning of epoch e_i we have, $\forall j \in [n]$, $\mathcal{Z}(i,k-1) \le c_1 n/A_{k-1} \cdot \omega(n)$, where $\omega(n)$ is a constant times $\log\log n$.

Inductive Step. Recall that a PrimaryJobQueue PJ_j is overloaded iff $c_1 n/A_k < \mathcal{Z}(i,k-1) \le c_1 n/A_{k-1} \cdot \omega(n)$. The amount $\mathcal{P}$ of pending threads during VS_k will have to be rescheduled among the remaining operational processors of $\mathcal{RM}$. In $\mathcal{P}$ we measure only assignments of threads to processors that remain live until their completions, because a thread assigned to a processor that dies before achieving its completion has already been included in $\mathcal{P}$, either as a pending thread because of a new death, or as a discarded thread at the beginning of VS_k.

Let f_k denote the set of newly dead processors during VS_k, and let D be the number of discarded threads at the beginning of VS_k. Then $F_k = |f_k| \leq rn$. Clearly, $\mathcal{P} = D + \sum_{i \in f_k} \mathcal{Z}_0(i,k) \leq a \cdot n$. Consider now a specific thread T_λ and suppose that the operational processors $P_{\lambda_1}, P_{\lambda_2}, \ldots, P_{\lambda_m}$ contest for it during a SECONDARYJOB. We already know that T_λ is covered by less that $c_3 \cdot \log n$ initially operational processors of $\mathcal{RM}$ with probability n^{-c_4}, for some positive constant c_3 and c_4 depending on c_3. So we can proceed with our proof conditioning on the event "$m \geq \log n$." By the rescheduling rule we have

$$\forall i \in [\log n], \qquad \mathbb{P}[P_{\lambda_i} \text{ gets } T_\lambda] \leq \frac{1/\mathcal{Z}_0(\lambda_i,k)}{\sum_{j=1}^{\log n}(1/\mathcal{Z}_0(\lambda_j,k))}. \tag{27}$$

It is now obvious that $\mathcal{Z}_0(\lambda_j,k) \leq c_1 n/A_k, \forall j \in [\log n]$. So, $\sum_{j=1}^{\log n}(1/\mathcal{Z}_0(\lambda_j,k)) \geq (A_k \cdot \log n)/c_1 n$ and thus we have

$$\forall i \in [\log n], \qquad \mathbb{P}[P_{\lambda_i} \text{ gets } T_\lambda] \leq \frac{c_1 n}{\mathcal{Z}_0(\lambda_i,k) \cdot \log n \cdot A_k}. \tag{28}$$

Now, each live processor P_i will randomly (with replacement) choose $\mathbf{b} \cdot \log\log n = c_1 n/A_k \cdot \log n \log\log n$ threads to cover during the SECONDARYJOBS of VS_k. Each random choice has a probability $\mathcal{P}/n$ of hitting a pending thread and is independent of the other choices. Thus the number of pending threads that P_i will contest for, is the number of successes from $\mathbf{b} \cdot \log\log n$ Bernoulli trials $B(q, \mathcal{P}/n)$, where $q \equiv c_1 n/A_k \cdot \log n \cdot \log\log n$. For each such thread, the probability of P_i prevailing over the other contestants (in a specific round of SDOU) has already been shown to be

$$\mathbb{P}[P_i \text{ gets } T_\lambda] \leq \frac{c_1 n}{A_k \cdot \log n} \cdot \frac{1}{\mathcal{Z}_0(\lambda_i,k)} \equiv \varphi.$$

Let Ξ_i be the number of pending threads that P_i contests for. Then, $\forall \varepsilon \in (0,1)$, $\Xi_i \in [(1-\varepsilon)\cdot q \cdot \mathcal{P}/n, (1+\varepsilon)\cdot q \cdot \mathcal{P}/n]$, with probability at least $1 - \exp(-\varepsilon^2/2 \cdot q \cdot \mathcal{P}/n)$, by a simple application of Chernoff Bounds on the Bernoulli trials. However, recall that $\mathcal{P} \leq a \cdot n$, and so, $\forall \varepsilon \in (0,1)$, $\Xi_i \in [(1-\varepsilon)\cdot q \cdot a, (1+\varepsilon)\cdot q \cdot a]$ with probability at least $1 - \exp(-\varepsilon^2/2 \cdot ac_1/(1-a) \cdot \log n \cdot \log\log n) = 1 - n^{-(\varepsilon^2 \cdot c_1 \cdot a \cdot \log e)/2(1-a) \cdot \log\log n})$. Now, given Ξ_i, due to our rescheduling rule, P_i will actually add at most $\Xi_i \cdot \varphi$ new threads in its own PrimaryJobQueue with probability at least $1 - n^{-c_9}$ as is easily shown by a new application of Chernoff Bounds, considering the worst case for P_i, $\mathcal{Z}(i,k) = 1$, for which φ is maximized. So, with total probability at most $(1-n^{-c_9}) \cdot (1-n^{-(\varepsilon^2 \cdot c_1 \cdot a \cdot \log e)/2(1-a) \cdot \log\log n})$ each operational processor P_i gets at most $\Xi_i \cdot \varphi$ threads, i.e.,

$$\Delta\mathcal{Z}(i,k) \leq (1+\varepsilon) \cdot a \cdot q \cdot \varphi \leq \frac{\beta}{\mathcal{Z}_0(i,k)}$$

with $\beta = (1+\varepsilon)a(c_1/(1-a))^2 \cdot \log\log n$. Hence, $\mathcal{Z}(i,k) = \mathcal{Z}_0(i,k) + \Delta\mathcal{Z}(i,k) = \mathcal{Z}_0(i,k) + \beta/\mathcal{Z}_0(i,k)$, which is maximized at $\max\{\mathcal{Z}(i,k)\} = \max\{1+\beta, 1+\gamma \cdot$

$\log\log n\} \cdot c_1 n/A_k$ and thus it satisfies the inductive hypothesis with probability of success at least

$$(1-n^{-\xi})^k(1-n^{-c_9})(1-n^{-(\varepsilon^2 c_1 a\cdot\log e)/2(1-a)\cdot\log\log n}) \geq (1-n^{-\xi})^{k+1},$$
$$\text{with} \quad \xi \leq \min\left\{c_9, \frac{\varepsilon^2 c_1 a\log e}{2(1-a)}\log\log n\right\}, \quad k \leq K_0. \qquad \square$$

Remark. Notice that $(1-n^{-\xi})^{k+1} \geq 1-(k+1)\cdot n^{-\xi}$, which implies that our robust system can tolerate computations of polynomial length.

5. Conclusions—Future Work

In this paper we have provided a general purpose simulation strategy for the execution of parallel algorithms in dynamically changing, decentralized computing environments. This simulation strategy is efficient in the sense that it imposes a polylogarithmic slowdown, compared with an execution of the input algorithm in a stable, totally reliable decentralized environment.

Our approach was based on three major axes: the provision of a robust storage scheme, the assurance (with high probability) of the even workload distribution among the live processing elements of the underlying machine, and the definite, epoch-by-epoch commitment of the work progress during the computation. Clearly, the latter technique was actually an attempt to compromise the heavy cost of assuring a definition progress (i.e., the completion of an epoch) by exploiting some intermediate tentative computations (i.e., the completion of a virtual superstep). The proposed simulation strategy, Robust-$\mathcal{BSP}$, is Las Vegas, since CHECKPOINT and BACKTRACK operations assure the work progress of the execution of the input algorithm, as long as there are at most $(1-a)n$ live processors during the whole process (a is an input parameter to our simulation strategy).

Yet, there remain several interesting open questions that arise through this work, which are also indicated in many other related articles in the literature, especially in the framework of load balancing. Such an open question might be the consideration of a dynamically changing (by means of unreliable nodes or links) network, that is continuously imposed computational threads, and the convergence of such a system to a stable state (if it ever converges).

Additionally, it would be very interesting to invent a deterministic simulation strategy that would be at least as efficient as Robust-$\mathcal{BSP}$. An intriguing open question is also a lower bound on the performance of any simulation strategy, that would depend on the fault occurrences during the execution of a BSP algorithm.

As for the kind of faults that are considered, in this work we studied the fail-stop model (restartable at any time, but reused from the first job after their reactivation). One can also extend our techniques to deal with malicious faults, an issue that is very challenging especially when dealing with decentralized computations over an insecure network infrastructure, such as the Internet.

References

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

[2] Y. Aumann, M. Bender, and L. Zhang. Efficient execution of non-deterministic parallel programs on asynchronous systems. In *Proc. of the* 8*th ACM Symposium on Parallel Algorithms and Architectures*, 1996, pp. 270–276.

[3] Y. Aumann, Z. Kedem, K. Palem, and M. Rabin. Highly efficient asynchronous execution of large-grained parallel programs. In *Proc. of the* 34*th Annual Symposium on Foundations of Computer Science*, 1993, pp. 271–280.

[4] A. Baumker, W. Dittrich, and F. meyer auf der Heide. Truly efficeint parallel algorithms: c-optimal multisearch for an extension of the *BSP* model. In *Proc. of the* 3*rd Annual European Symposium on Algorithms*, LNCS 979, Springer-Verlag, Berlin, 1995, pp. 17–30.

[5] P. Berenbrink, F. Meyer auf der Heide, and V. Stemann. Fault-tolerant shared memory simulations. In *Proc. of the* 13*th Annual Symposium on Theoretical Aspects of Computer Science* (*STACS* '96), Springer-Verlag, Berlin, 1996, pp. 181–192.

[6] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. of the* 35*th Annual Symposium on Foundations of Computer Science*, 1994, pp. 356–368.

[7] B. Chlebus, A. Gambin, and P. Indyk. PRAM computations resilient to memory faults. In *Proc. of the* 2*nd Annual European Symposium on Algorithms* (*ESA* '94), LNCS 855, Springer-Verlag, Berlin, 1994, pp. 401–412.

[8] B. Chlebus, A. Gambin, and P. Indyk. Shared-memory simulations on a faulty DMM. In *Proc. of the International Colloquium on Automata, Languages and Programming*, 1996, pp. 586–597.

[9] B. Chlebus, L. Gasieniec, and A. Pelc. Fast deterministic simulation of computations on faulty parallel machines. In *Proc. of the* 3*rd Annual European Symposium on Algorithms*, LNCS 979, Springer-Verlag, Berlin, 1995, pp. 89–101.

[10] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[11] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: a fault tolerant, high performance approach. In *Proc. of the* 15*th International Conference on Distributed Systems*, 1995, pp. 467–474.

[12] R. De Prisco, A. Mayer, and M. Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proc. of the ACM Symposium of Distributed Computing*, 1994, pp. 161–171.

[13] D. Dubhashi, V. Priebe, and D. Ranjan. Negative Dependence through the FKG Inequality. BRICS Report Series, RS-96-27, ISSN 0909-0878.

[14] D. Dubhashi and D. Ranjan. Balls and Bins: a Study in Negative Dependence. BRICS Report Series, RS-96-25, ISSN 0909-0878.

[15] C. Dwork, J. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. In *Proc. of the* 11*th ACM Symposium on Principles of Distributed Computing*, 1992, pp. 91–102.

[16] Z. Galil, A. Mayer, and M. Yung. Resolving message complexity of Byzantine agreement and beyond. In *Proc. of the* 36*th IEEE Annual symposium on Foundations of Computer Science*, 1995, pp. 724–733.

[17] J. Garofalakis, S. Rajsbaum, P. Spirakis, and B. Tampakas. Tentative and definite distributed computations: an optimistic approach to network synchronization. *Journal of Theoretical Computer Science*, 128(1–2):63–74, 1994.

[18] A. Gerbessiotis and C. Siniolakis. Communication Efficient Data Structures on the *BSP* Model with Applications. Technical Report PRG-TR-13-96, Oxford University, May 1996.

[19] A. Gerbessiotis and L. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

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

[21] A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bound for occupancy and the satisfiability threshold conjecture. In *Proc. of the* 35*th IEEE Annual Symposium on Foundations of Computer Science*, 1994, pp. 592–603.

[22] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms on restartable fail-stop processors. In *Proc. of the* 10*th Annual ACM Symposium on Principles of Distributed Computing*, 1991, pp. 23–36.

[23] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5:201–217, 1992.

[24] P. Kanellakis and A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic, Dordrecht, ISBN 0-7923-992-6, 1997.

[25] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proc. of the* 23*rd Annual ACM Symposium on Theory of Computing*, 1991, pp. 381–390.

[26] Z. Kedem, K. Palem and P. Spirakis. Efficient robust parallel computations. In *Proc. of the* 22*nd Annual ACM Symposium on Theory of Computing*, 1990, pp. 138–148.

[27] S. Kontogiannis, G. Pantziou, and P. Spirakis. Efficient computations on fault-prone BSP machines. In *Proc. of the* 9*th ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 84–93. Full version at `http://www.ceid.upatras.gr/~kontog/bsp/spaa97/`

[28] S. Kontogiannis, G. Pantziou, P. Spirakis, and M. Yung. Dynamic-fault-prone BSP: a paradigm for robust computations in changing environments. In *Proc. of the* 10*th ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, June 28–July 2, 1998, pp. 37–46. Full version at `http://www.ceid/upatras.gr/~kontog/bsp/spaa98/`

[29] F. T. Leighton, B. M. Maggs, and R. K. Sitaraman. On the fault tolerance of some popular bounded-degree networks. *SIAM Journal on Computing*, 27(5):1303–1333, 1998.

[30] W. F. McColl. Scaleable parallel computing: a grand unified theory and its practical development. In *Proc. of IFIP World Congress*, Hamburg, August, 1994, Vol. 1, pp. 539–546.

[31] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, 1995.

[32] M. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2): 335–348, April 1989.

[33] M. Sipser and D. Spielman. Expander codes. In *Proc. of* 35*th Annual Symposium on Foundations of Computer Science*, 1994, pp. 566–576.

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

[35] L. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science*, J. van Leeuwen ed. North-Holland, Amsterdam, 1990, pp. 943–972.

Online publication October 25, 2000.