Fully-Scalable Fault-Tolerant Simulations for BSP and CGM

Sung-Ryul Kim
kimsr@algo.snu.ac.kr

Kunsoo Park
kpark@theory.snu.ac.kr

Department of Computer Engineering
Seoul National University
Seoul 151-742, Korea

Abstract

In this paper we consider general simulations of algorithms designed for fully operational BSP and CGM machines on machines with faulty processors. The faults are deterministic (i.e., worst-case distributions of faults are considered) and static (i.e., they do not change in the course of computation). We assume that a constant fraction of processors are faulty. We present a deterministic simulation (resp. a randomized simulation) that achieves constant slowdown per local computations and $O((\log_h p)^2)$ *(resp.* $O(\log_h p)$*) slowdown per communication round, provided that a deterministic preprocessing is done that requires* $O((\log_h p)^2)$ *communication rounds and linear (in h) computation per processor in each communication round. Our results are fully-scalable over all values of* p *from* $\Theta(1)$ *to* $\Theta(n)$*. Furthermore, our results imply that for* $p \leq n^\epsilon$ *(*$\epsilon < 1$*), algorithms can be made resilient to a constant fraction of processor faults* without *any asymptotic slowdown.*

1 Introduction

Most of theoretical research on parallel computation has focused on shared-memory models such as PRAM [15] or specific interconnection network models such as the hypercube [23]. In these models the ratio of memory to processors is fairly small (typically $O(1)$). Recently, bridging models such as BSP [30, 31], CGM [7], and LogP [6, 18] have been proposed, where the ratio of memory to processors is non-constant. In particular, the BSP model attracted considerable attention and many algorithms for important problems have been designed on this model [12, 13, 14, 8]. In BSP and CGM there are p processors and a memory of n words (usually, n is also the input size) is distributed evenly across the processors.

The BSP and CGM models, as is usually the case, refers to ideal machines where all processors are fully operational. But the increasing complexity of multiprocessor computers makes the machines prone to hardware failures. This necessitates the design of algorithms that are resilient to hardware faults. A most natural solution is to design a general simulation mechanism of an ideal machine on its faulty counterpart.

In this paper we develop general techniques that can efficiently simulate ideal BSP (CGM) computations on BSP (CGM) machines where some processors may be faulty. The faults are deterministic (i.e., worst-case distributions of faults are considered) and static (i.e., they do not change in the course of computation). We assume that a constant fraction of processors are faulty and that a processor that tries to communicate with a faulty processor is notified, at the end of the communication round, that the communication was unsuccessful.

1.1 The BSP and CGM models of computation

We first describe the BSP model. CGM is different from BSP only in the way the cost of a computation is calculated. A BSP machine consists of p processor/memory components communicating through some interconnection network. L is the time for a global barrier synchronization and g describes the ratio of computation and communication throughput. The BSP machine proceeds in a sequence of communication/computation rounds (which Valiant calls supersteps). In a single superstep each processor may send or receive h messages (typically, $h = \Theta(n/p)$ where n is the total memory size) and then perform an internal computation on its internal memory. The parameter h is called the *bandwidth* a processor uses. The internal computation and the messages sent out in a round t can depend only on data locally available before the start of round t (i.e., on messages received in round $t-1$, but not on messages received in round t).

Each superstep is charged a cost of $\max\{L, x, hg\}$ where x is the maximal number of local computations on every processor and h is the maximal number of packets that any processor sends or receives in the superstep. The total cost of an algorithm is the sum of the costs of each superstep performed by the algorithm.

The CGM model explicitly specifies the parameter h, i.e., it is assumed that $h = \Theta(n/p)$. The objective of designing an efficient CGM algorithm is to minimize the local computation time and the number of communication rounds.

It is convenient to assume a global address space containing the memory of all processors. A memory location with address l in $P(i)$ is represented globally by the pair (i, l). For a datum d that is stored in global address (i, l), $Addr(d)$ (the *pointer* to d) denotes the global address (i, l).

1.2 Related works

The fault-tolerant simulation for PRAM with faulty processors was first studied by Kanellakis and Shvartsman [17] (for the deterministic distribution of faults) and Kedem, Palem and Spirakis [20] (for probabilistic faults). They considered *fail-stop dynamic faults*, i.e., faults may happen at any time during the computation and cause processors to stop till the end of computation. For the same model of faults Kedem et al. [19] designed fault-tolerant simulations that have constant expected slowdown under some assumptions on the faults probability. Diks and Pelc [9] developed efficient simulation algorithms for EREW PRAM under the probabilistic processor-fault model. Kanellakis, Michailidis and Shvartsman [16] developed an optimal simulation for EREW PRAM when all processor faults are static. Buss et al. [2] developed deterministic PRAM simulations under the model of restartable failures.

For PRAM with faulty memory, Chlebus, Gambin and Indyk [3] designed randomized simulations for both dynamic and static faults, assuming a constant fraction of deterministic memory faults. For static faults, they gave a simulation algorithm of n-processor CRCW PRAM on $n/\log n$-processor CRCW PRAM with $O(\log n)$ step overhead, which is preceded by an $O(\log^{3/2} n)$-time preprocessing. They also gave an algorithm for performing the simulation in real time, but in this case the number of processors of the faulty machine grows to $O(n \log n)$. Chlebus, Gąsieniec and Pelc [5] presented a deterministic simulation of n-processor EREW PRAM on n-processor EREW PRAM prone to processor and memory failures, with step overhead $O(\log n)$ and preprocessing time $O(\log^2 n)$. Using randomization, Gąsieniec and Indyk [10] reduced the number of processors to $n/\log n$ and the preprocessing time to $O(\log n)$ without changing the step overhead, hence making the simulation work-optimal. They also proved the existence of a deterministic algorithm with the same step overhead and preprocessing time. Recently, Chlebus, Gambin and Indyk [4] and Berenbrink, Meyer auf der Heide and Stemann [1] obtained several simulations of EREW PRAM on DMM for static and dynamic models of faults.

Currently, the best simulation techniques for BSP are due to Kontogiannis, Pantziou, Spirakis, and Yung [21, 22]. They considered both static and dynamic processor faults. For static faults, [21] gives a randomized technique that has constant slowdown for local computations and $O(F(p))$ slowdown per communication round, provided a preprocessing is done that requires $O(p/F(p) + \log^2 p)$ communication rounds, where $F(p)$ can have any value between 1 and p. Additionally, $\Theta(p/F(p))$ memory is needed per processor. For dynamic faults the best result [22] is a randomized technique that has a competitive ratio of $O((\log p \log\log p)^2)$ against an optimal off-line strategy. To our best knowledge, there are no previous works on the fault-tolerance of CGM.

1.3 Our Results

We present general fault-tolerant simulations of computations on BSP and CGM models under static processor faults. We say that a time (or space) bound, which depends on a number n, holds *with high probability* if the probability that the bound will hold is at least $1 - n^{-\alpha}$, where α can be any positive constant depending on the constant factor of the bound.

R1: A p-processor BSP and CGM machine with a constant fraction of faulty processors can *deterministically* simulate its fault-free counterpart with constant slowdown for local computations and $O((\log_{\bar{h}} p)^2)$ slowdown per communication round, provided that a preprocessing is done that requires $O((\log_{\bar{h}} p)^2)$ communication rounds and linear (in $\bar{h}$) computation per processor in each communication round.

R2: A p-processor BSP and CGM machine with a constant fraction of faulty processors can simulate its fault-free counterpart with constant slowdown for local computations and $O(\log_{\bar{h}} p)$ slowdown per communication round with high probability, after the same preprocessing as in R1.

The parameter $\bar{h}$ above is the bandwidth of the simulator, which is set to n/p on CGM where only the number of communication rounds is important. On BSP, the average bandwidth h_{avg} of the simulated algorithm is assumed to be given to the simulator. We will later explicitly consider how $\bar{h}$ can be selected on BSP to give a minimal slowdown.

Note that the slowdowns that our results achieve are significantly better than the previous ones for BSP [21, 22].

Further, one of our results is a deterministic one, while the previous works are all randomized ones.

In BSP and CGM algorithms, *scalability* is an important issue since large scalability enables algorithms to be applicable without modifications to a wide range of parallel machines while retaining the efficiency. We explicitly consider the memory requirements of the simulation techniques and our simulation techniques work with asymptotically the same amount of memory as that of the simulated algorithm, even when there are only constant amount of memory available per processor. Hence, our results are fully-scalable over all values of p from $\Theta(1)$ to $\Theta(n)$.

Another issue that should be addressed in fault-tolerance is that of input reconstruction (i.e., recovering the lost parts of the input by decoding). We can modify the technique in [5] so that the input reconstruction can be performed efficiently on BSP and CGM models, but we omit the details in this version.

We now consider the implications of our simulation techniques in practical BSP and CGM computations. In almost all practical BSP and CGM computations, $p \leq n^{\epsilon}$ for some constant $0 < \epsilon < 1$. Also, nearly all efficient BSP algorithms have $h = \Theta((n/p)^{\delta})$ for some constant $\delta > 0$. Hence, the preprocessing and communication slowdown of our techniques turn out to be constants in these practical cases. That is, fault-tolerance incurs *no asymptotic slowdown in practical cases* with our simulation techniques. This implies that efficient BSP and CGM algorithms for sorting [13], list ranking [8], convex hull [14], and so on [14, 12] can be made resilient to a constant fraction of processor faults without any slowdown even when worst-case fault distributions are assumed. To our knowledge, no previous fault-tolerance results on any general model of computation have this strong property.

2 Algorithm overview

We first describe our simulations for the CGM model where $\bar{h} = h = \Theta(n/p)$. The modifications needed to make the simulations fully general on BSP will be described at the end of this paper. Our simulation for CGM consists of two parts: the preprocessing and the actual superstep-by-superstep simulation of the simulated algorithm. Let $P(i)$, $1 \leq i \leq p$, denote the i-th processor in an ideal CGM machine. The number i will be called the *identification* of $P(i)$. If $P(i)$ is not faulty, we say that $P(i)$ is *active*. For simplicity, we assume that all divisions, square-roots, and logarithms we use give integers.

In the preprocessing we first identify the number of active processors among $P(i)$, $1 \leq i \leq p$. If there are q active processors in total, we will assign unique numbers (*ranks*) from 1 to q to them. We assume that $q/p \geq C$, where C, $0 < C < 1$, is a constant parameter given to the simulator. Let $Q(j)$, $1 \leq j \leq q$, denote the active processor to which the rank j is assigned.

Once the ranks of all active processors are known, we can simulate the local computations easily. $Q(i)$ simulates the computations of $P(i\frac{p}{q} - \frac{p}{q} + 1)$ through $P(i\frac{p}{q})$. Since $q \geq Cp$, the number of ideal processors that is simulated by $Q(i)$ is at most $1/C$, which is a constant. Hence the local computations can be simulated with a constant slowdown.

After the local computations are complete, each active processor $Q(i)$ has at most h/C messages destined for some other processors (called the *targets* of the messages). The problem is that $Q(i)$ does not know which processors are simulating the targets of the messages. For example, consider the following situation. One of the processors that $Q(i)$ is simulating is $P(j)$ and $P(j)$ has a message to be sent to $P(k)$. Assume that $P(k)$ is being simulated by an active processor $Q(k')$. $Q(i)$ can easily compute k' from k. To communicate using the underlying communication network, however, $Q(i)$ should know the identification of $Q(k')$. Since faults may occur in an arbitrary pattern and the amount of memory that $Q(i)$ has may be much smaller than q words, $Q(i)$ cannot store the identifications of all the active processors. Hence, we need a way of somehow *routing* the messages through a structure built among the active processors. For the purpose of routing the messages, we use the *generalized butterfly* [23]. Our two simulation techniques differ only in the ways the messages are routed. Both of them use the same preprocessing.

3 Preprocessing

We describe the preprocessing that requires $O((\log_h p)^2)$ communication rounds. We say that a group of processors is *good* if the ratio of active processors in the group is at least C. A good group of processors that has b active processors in total is *ranked* if b is known by all the active processors in the group and each active processor is assigned a unique rank in $\{1, \ldots, b\}$. The preprocessing consists of two parts: ranking $\{P(1), \ldots, P(p)\}$ and building a $\sqrt{h}$-ary generalized butterfly with the active processors.

3.1 Ranking

The ranking is performed by constructing an h-ary tree consisting of all the active processors in $\{P(1), \ldots, P(p)\}$. The ranking consists of $\log_h p$ phases.

The following notations are used for trees. Let T be a tree. $Root(T)$ denotes the root node of the tree. $Level(u)$ of a node u in T denotes the length of the path (the number of the edges in the path) from $Root(T)$ to u. $Height(T)$ denotes the maximum of $Level(u)$ when u ranges over all nodes in T.

Let a *group* $G(i,k)$ ($0 < i \leq \log_h p$, $0 < k \leq p/h^i$) denote the set of h^i processors $\{P(l) : h^i(k-1) < l \leq h^i k\}$. In the i-th phase, an h-ary tree consisting of the active processors in each good group $G(i,k)$ is constructed. The i-th phase consists of $O(\log_h h^i) = O(\log_h p)$ communication rounds.

We describe the first phase. Fix one $G(1,k)$. There are h processors in $G(1,k)$. Each active processor in $G(1,k)$ sends a query message to all other processors in $G(1,k)$. Then all the active processors in a group know which processors in the group are active. An active processor (say, one with the largest identification) is selected as the leader of a group. If a group is good, the leader of the group forms an h-ary tree of height 1 such that the leader is the root and all other processors of the group are the leaves, and ranks all the processors such that the leader has the largest rank.

We now state the phase invariants after the i-th phase.

1. The active processors of a good group $G(i,k)$, $0 < k \leq p/h^i$, form an h-ary tree T of height $O(\log_h h^i) = O(i)$.
2. Each good group $G(i,k)$, $0 < k \leq p/h^i$, is ranked.

Note that the phase invariants are satisfied after the first phase. A general i-th phase consists of two subphases: tree-construction and ranking subphases.

3.1.1 The tree-construction subphase

For simplicity we describe the computation in $G(i,1)$ Note that $G(i,1)$ is the union of $G(i-1,1), \ldots, G(i-1,h)$. Each $G(i-1,j)$ is called a *subgroup* of $G(i-1,1)$. The tree-construction subphase consists of the following steps.

1. Each good subgroup $G(i-1,j)$, $0 < j \leq h$, identifies all the active processors in $G(i,1)$. Each active processor $P(k)$ in $G(i-1,j)$ sends query messages to $O(h)$ processors (allocated to $P(k)$ according to the rank of $P(k)$ in $G(i-1,j)$) in $G(i,1)$ and stores the identifications of the processors found to be active.
2. Among $G(i-1,1), \ldots, G(i-1,h)$, let $G(i-1,l)$ be the good subgroup having maximal l. Even if there are several good subgroups, $G(i-1,l)$ can be easily selected since all the processors in any subgroup receives at least one message from a processor in each good subgroup in step 1. Let T denote the h-ary tree consisting of the active processors in $G(i-1,l)$.
3. The processors in $G(i-1,l)$ adjusts T to form an h-ary tree T' with all the active processors in $G(i,1)$. The details are omitted.

It can be shown that the above procedure is implemented on CGM in $O(\log_h q)$ communication rounds and that $Height(T')$ is $O(i)$.

3.1.2 The ranking subphase

The total number of active processors is counted by ranking all the processors, using a prefix-sums computation on T' with all the processors in T' initially holding a 1, such that the ordering of the nodes is that of the postorder traversal of T'. The processor $Root(T')$ checks if $G(i,1)$ is good and informs all the active processors in $G(i,1)$ of the result. It is clear that $G(i,1)$ satisfies invariant 1.

We can easily verify that our ranking algorithm ranks $\{P(1), \ldots, P(p)\}$ and forms an h-ary tree of height $O(\log_h p)$ with all the active processors. Since $\{P(1), \ldots, P(p)\}$ is good, one of its h subgroups must be good, and the property must hold recursively in the good subgroup. Call the final tree T.

Property 1 *T satisfies the following by the ranking subphase.*

1. *$Root(T)$ has the highest rank among the processors in T.*
2. *A subtree rooted at a child of $Root(T)$ contains processors with consecutive ranks.*
3. *Properties 1 and 2 hold recursively in each subtree.*

3.2 Building a generalized butterfly

We build a generalized butterfly using the active processors. The generalized butterfly we will build is a $(2\log_h q)$-column wrapped butterfly consisting of q processors where each processor has $2\sqrt{h}$ connections. Here, we say that there is a connection between $Q(i)$ and $Q(j)$ when $Q(i)$ knows the identification of $Q(j)$ and vice versa.

Imagine the processors $Q(i)$, $1 \leq i \leq q$, are placed in a $q/(2\log_h q)$ by $2\log_h q$ grid in the column major order. Throughout the paper, r and c denote $q/(2\log_h q)$ (the number of rows) and $2\log_h q$ (the number of columns), respectively. Also, $B(i,j)$, $0 \leq i < r$ and $0 \leq j < c$, denotes the processor at the (i,j) position of the grid, i.e., $B(i,j) = Q(jr+i+1)$. The processors in column j (row i) are collectively denoted by $col(j)$ ($row(i)$). Let *block* k $b(x,k)$ of a number x that is represented in d bits, where $0 \leq k < 2\log_h d$, refer to the bits from the $(\frac{k\log h}{2}+1)$-st least significant to the $(\frac{(k+1)\log h}{2})$-th least significant in the bit representation of x. In the generalized butterfly we construct, $B(i,j)$ and $B(i',(j+1) \bmod c)$ (henceforth we will omit all modular operations) are connected if $i = i'$ or only $b(i,j)$ and $b(i',j)$ are different in the bit representation of i and i'. A connection between $B(i,j)$ and $B(i',j+1)$ is called a *row connection* if $i = i'$.

Now we describe how to form a generalized butterfly network with the active processors. All the trees we will construct from T satisfy Property 1. We will use a procedure that splits a tree U satisfying Property 1 into k

trees $U_0, \dots, U_{k-1}$ so that each U_i, $0 \le i < k$, contains processors with consecutive ranks and satisfies Property 1. The procedure can be easily implemented by generalizing the tree splitting technique in [28] so that the number of communication rounds required is proportional to the height of U and each processor performs communication and local computation proportional to h in each communication round, regardless of k. Also, we can implement the procedure so that each $Root(U_i)$ knows the identifications of $Root(U_{i\pm1})$. The details are omitted. From now on, *"evenly split a tree"* means the application of this procedure.

3.2.1 Split into columns

Evenly split T into c trees $T_0, T_1, \dots, T_{c-1}$ (so that $col(j)$ are in T_j). Note that $Root(T_j)$, $0 \le j < c$, knows the identifications of $Root(T_{j\pm1})$. This requires $O(\log_h q)$ communication rounds since $Height(T)$ is $O(\log_h q)$.

3.2.2 Build row connections

Let F denote the forest consisting of T_j, $0 \le j < c$. We will repeatedly split a forest into several forests (i.e., split the trees in a forest and combine resulting trees into forests) while maintaining the following invariant, which is satisfied by F. Given a tree T consisting of active processors, the rank of a processor $Q(i)$ *within* T is defined to be the number of processors in T having smaller ranks than that of $Q(i)$.

Invariant 1

1. *The processors of each $row(a)$, $0 \le a < r$, are in one forest.*
2. *There is exactly one processor of each $row(a)$ in one tree of a forest and the processors of one $row(a)$ have the same ranks within the trees they are in.*

In $O(\log_h q)$ communication rounds, we evenly split each T_j, $0 \le j < c$, into h trees T_{ij}, $0 \le i < h$, and propagate all the identifications of $Root(T_{ij})$ to $Root(T_j)$ using the circular list among $Root(T_{ij})$'s. Using the connection between $Root(T_j)$ and $Root(T_{j+1})$, each pair $Root(T_{ij})$ and $Root(T_{i,j+1})$, $0 \le i < h$, are connected in a constant number of communication rounds. Similarly, each pair $Root(T_{ij})$ and $Root(T_{i,j-1})$, $0 \le i < h$, are connected. Hence, T_{ij}, $0 \le j < c$, forms a forest, denoted by F_i. The invariants are clearly satisfied since T_j's are split in the same way.

We can repeat the above until each tree contains only one processor and can no longer be split ($O(\log_h q)$) repetitions). Then, each forest will contain the processors in one row by the invariant and the row connections can be easily built. We have built the row connections in $O((\log_h q)^2)$ communication rounds.

3.2.3 Build the remaining connections

To build the remaining connections, we first connect, in each $col(j)$, $0 \le j < c$, all the pairs of processors $B(i,j)$ and $B(i',j)$ where the bit representations of i and i' differ only in $b(i,j)$ and $b(i',j)$. (Such connections are called *generalized hypercube connections*.) Actual butterfly connections (i.e., between $col(j)$ and $col(j+1)$) can be built later using the row connections.

First note that each T_j constructed above contains the processors in $col(j)$. T_j is evenly split so that each resulting tree contains $(\sqrt{h})^{j+1}$ processors.

Call one of the resulting trees U. All generalized hypercube connections to be made are within U. We first split U into $\sqrt{h}$ trees $U_0, \dots, U_{\sqrt{h}-1}$ that form a forest F'. As in 3.2.2 we keep splitting F' into several forests while keeping the set of processors to be connected by generalized hypercube connections in one forest. The details are omitted.

4 Superstep-by-superstep simulation

We already mentioned that the simulation of local computations is easy. The simulation of communications in one superstep can be considered to be a routing problem in the generalized butterfly where each processor sends and receives h/C messages. We present two routing techniques. One is a deterministic technique and the other is a randomized one. For simplicity, we assume that an active processor simulates one ideal processor rather than $1/C$ ideal processors (i.e., at most h messages are sent and received by a processor) because the extension to $1/C$ ideal processors is trivial and incurs only constant slowdown. We first describe how to route a single message in the generalized butterfly.

4.1 Routing in the generalized butterfly

We describe the way of using the generalized butterfly to route the messages by showing how *one* message m originating from an arbitrary processor $B(i,j)$ can be routed to another arbitrary processor $B(i',j')$. Let $TRA(m)$ denote the *target row address* of m (i' in this case). Let $B(x,y;k)$ denote the processor $B(x', y+1)$ where x' differs from x only in block y and $b(x',y) = k$. The routing consists of three stages each of which consists of at most $\log_h q$ communication rounds.

S1: In each communication round, m moves one column to the right using the row connections until m wraps around to $B(i,0)$.

S2: This stage consists of c *routing rounds* numbered from 0 to $c-1$. Let i_0 be i. In routing round k, message m is moved from $B(i_k, j)$ to processor $B(i_{k+1}, j+1)$, where $b(i_{k+1}, j) = b(TRA(m), j)$ (i.e., $B(i_{k+1}, j+1) = B(i_k, j; b(TRA(m), j))$).

S3: In each communication round, m moves one column to the right until m reaches $B(i',j')$.

4.2 Deterministic routing

Butterfly routing does not give a good deterministic bound because of a well-known lower bound [23], which implies that there are cases where the number of messages that pass through a processor in S2 is as large as $\sqrt{hq}$ if we route all the messages to their respective target rows.

In our situation, however, a processor $Q(i)$ can communicate directly with any processor whose identification is known to $Q(i)$, and thus the messages do not have to really *pass through* a processor, and it is possible to efficiently route a large number of messages through a processor. The idea is to form a tree (called a *distance tree*) containing the set of messages that pass through a processor and route only the root of the tree through the processor. Trees must be split and merged because a processor is connected to several processors. A similar idea was used by Chlebus et al. [5] in a different context (in a linked list). We need a nontrivial generalization to the generalized butterfly because of load balancing among the processors.

4.2.1 High-level description of deterministic routing

We give a high-level description without giving the details of the distance tree (i.e., the structure of the tree and how the nodes of the trees are distribued among active processors) we will use. We only mention that $D(i,j)$ uses a memory linear in the number of messages in $M(i,j)$. For the time being, we will focus on the messages that originate from $col(0)$. The processors in $col(0)$ will be called the *owners* and all the processors (including the owners) will be called *routers* when they are used to identify the path that messages should follow in S2. The messages originating from other columns will be considered later. We use the following notations. In S2, for a router $B(i,j)$, $j \neq 0$, the set of messages that come into $B(i,j)$ and the set of messages go out of $B(i,j)$ are the same. But for $B(i,0)$ the two sets are not the same. Let $M(i,j)$ denote the set of messages that is routed *out of* $B(i,j)$ in stage S2 of the butterfly routing. For $B(i,0)$, the set of messages that come into $B(i,0)$ as the result of S2 is denoted by $M(i,c)$. $M(i,j;k)$ denotes the subset of $M(i,j)$ that is routed to $B(i,j;k)$. $D(i,j)$ and $D(i,j;k)$ denote the distance tree containing the messages in $M(i,j)$ and $M(i,j;k)$, respectively.

We describe the deterministic routing stage by stage. First, S1 is not needed for messages originating from $col(0)$ since the messages are initially in $col(0)$.

Deterministic implementation of S2 consists of the implementation of c routing rounds. Initially, each $D(i,0)$ is constructed in $B(i,0)$. We maintain an invariant that the processor that holds each $Root(D(i,j))$ knows the identification of $B(i,j)$, which is initially satisfied since each $B(i,0)$ holds the entire $D(i,0)$. Each routing round j consists of two parts. In the first part each $D(i,j)$ is split into $D(i,j;0),\ldots,D(i,j;\sqrt{h}-1)$ and each $Root(i,j;k)$ is sent to $B(i,j;k)$ (the identification can be obtained from $B(i,j)$). In the second part, called the *merge* part, the (splited) distance trees that are routed to $B(i,j+1)$ are merged into $D(i,j+1)$. We mention that $B(i,j+1)$ participates in the construction of $Root(D(i,j+1))$ so that the above mentioned invariant is satisfied. After all the routing rounds are complete, we will have $D(i,c)$'s for each i and S2 is complete.

For S3, fix a row i. By the above mentioned invariant, the processor $Q(s)$ holding $Root(D(i,c))$ knows the identification of $B(i,0)$. The deterministic implementation of S3 consists of c rounds numbered from 0 to $c-1$. In round j, the identification of $B(i,j)$ is broadcast (through the tree structure of $D(i,c)$) to all the processors holding the messages in $D(i,c)$. Then, the messages in $M(i,c)$ whose target is $B(i,j)$ is sent to $B(i,j)$ using the broadcast identification. For the next round, $Q(s)$ obtains the identification of $B(i,j+1)$ using the row connection of $B(i,j)$.

4.2.2 Distance tree

The operations we use on a distance tree $D(i,j)$ are as follows. Initially, we sequentially construct the distance trees $D(i,0)$, $0 \leq i < r$. In S2, we split $D(i,j)$ into $D(i,j;k)$'s and merge at most $\sqrt{h}$ distance trees into one distance tree. In S3, we broadcast a value to the processors holding the nodes of the distance tree. A data structure that can facilitate the four operations while using only linear memory is the *compacted trie*.

The distance tree $D(i,j)$ is defined to be a compacted trie containing the messages in $M(i,j)$. By defining the key (used in the distance tree) of each message m in $M(i,j)$, we can implement the four operations efficiently. Let $TRA'(m)$ be $Addr(m) \cdot r + TRA(m)$. The key of a message m in $D(i,j)$ is defined to be the sequence of blocks $b(TRA'(m),j), b(TRA'(m),j),\ldots,$ $b(TRA'(m), 2(\log_h n + log_h r))$. We use $TRA'(m)$ rather than $TRA(m)$ because the key of a message m must be unique in a trie. Note that the key of a message m can be represented using a constant number of words.

Since the number of messages in $D(i,j)$ can be proportional to $\sqrt{hq}$, which can be larger than $O(n/q)$, the memory each processor has, the nodes of $D(i,j)$ are distributed among the owners with the restriction that one node is stored in one processor.

See Fig. 1 for an example distance tree at a router in column 3. Assume that the edge from u to v contains two blocks of key. The messages m in the subtree under the

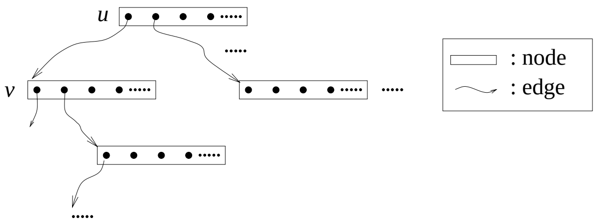

Figure 1. An example distance tree

node v shares blocks 3 and 4 of $TRA(m)$ and hence follow the same path until column 5 and are routed to separate routers at column 6 when they leave column 5.

4.2.3 Implementations of the operations

The detailed implementations of the four operations on CGM are omitted due to space limitations. The sequential construction of the distance tree can be done with $O(h \log_h q)$ computation and no communication. The split uses only the root of the distance tree and it requires $O(\sqrt{h})$ computation and no communication. The merge operation takes $O((\log_h q)^2)$ communication rounds with $O(h)$ computation per processor in each communication round. The broadcast takes $O(\log_h q)$ communication rounds with $O(h)$ computation per processor in each communication round.

Of the four operations, the only difficult operation to implement is the merge operation, which includes a nontrivial load balancing procedure. We just mention that in order to construct each distance tree $D(i,j)$, $0 \leq i < r$, the merge is performed level-by-level from the top level of $D(i,j)$ using only two levels at a time.

4.2.4 Pipelining and the cost of deterministic routing

As described, the routing of the messages originating from all c columns takes $O((\log_h q)^4)$ communication rounds since stage S2 of the routing of the messages originating from one column takes $O((\log_h q)^3)$ communication rounds. We use pipelining twice to reduce the number of communication rounds to $O((\log_h q)^2)$.

We first pipeline the merge operations in different routing rounds. We can split a distance tree $D(i,j)$ immediately after the root is constructed because the split operation uses only the root of a distance tree. The merge construction of all $D(i,j+1)$, $0 \leq i < r$, simultaneously starts after the top 3 levels of $D(i,j)$, $0 \leq i < r$, are constructed. Since the implementation of the merge operation uses only two levels at a time, the merges in different routing rounds do not interfere with each other.

We next pipeline the routing of messages originating from all columns. First, the purpose of S1 is to reach $col(0)$ so it can be replaced by the broadcast of the identification of $B(i,0)$ in each $row(i)$. For S2, processors in $col(j)$ starts S2 by performing routing round 0 when the processors in $col(0)$ performs routing round j. In the same way, processors in $col(j)$ starts S3 when processors in $col(0)$ performs round j of S3. We have shown the result R1 for CGM.

4.3 Randomized routing

When all the messages are routed simultaneously, there is a well-known randomized solution that requires asymptotically the same number of communication rounds as when only one-message is routed. Specifically, we use the following result due to Ranade [24, 25, 26, 27, 23].

Theorem 1 *[23] Given any routing problem on a generalized butterfly where the maximum number of messages passing through one connection is K, Ranade's algorithm will move every packet to its destination in $O(\log_h q + K + \log q)$ constant-message communication rounds with high probability using a constant number of buffers per connection.*

Using the two-phase routing strategy of Valiant [29, 32, 23], we can obtain an upper bound on K in the above theorem that holds with high probability. Specifically, we can bound K and the number of constant-message communication rounds in Theorem 1 by $O(\sqrt{h} \log_h q)$. Detailed analysis is omitted.

Since we can use $O(\sqrt{h})$ messages per communication round and $O(\sqrt{h})$ buffer for one connection, we can pipeline the routing so that the number of communication rounds needed are reduced by a factor of $\sqrt{h}$. Thus we have the result R2 for CGM.

5 Simulations for BSP

There are problems in applying the CGM simulation techniques to BSP since the bandwidth h, which is fixed to be $\Theta(n/q)$ in CGM, may change from superstep to superstep in BSP. Although most BSP algorithms use fixed values for h, they may not be $\Theta(n/q)$. The bandwidth h can even be larger than n/q, the memory each processor has [11]. Hence, the first problem is how to set the bandwidth $\bar{h}$ in the BSP simulation.

We assume that h_{avg}, which is the average bandwidth of all the supersteps in the simulated algorithm, is given to the simulation. We will set $\bar{h}$ as large as possible according to the relationship among the values h_{avg}, L/g, and n/q. Note that n/q is an upper bound on $\bar{h}$ since we use $O(\bar{h})$ memory per processor. There are three cases. First, if $h_{avg} < \min\{L/g, n/q\}$, then we set $\bar{h}$ as $\min\{L/g, n/q\}$ because (in this case) a superstep with bandwidth h_{avg} will have the same cost as that with bandwidth $\min\{L/g, n/q\}$.

Second, if $L/g \leq h_{avg} < n/q$, then we set $\bar{h}$ as h_{avg}. Finally, if $h_{avg} > n/q$ we have to set $\bar{h}$ as n/q.

The preprocessing of the BSP simulation is the same as that of the CGM simulation. The preprocessing overhead will be $O((\log_{\bar{h}} p)^2)$ communication rounds with bandwidth $O(\bar{h})$. Also, the local computations can be simulated with constant slowdown as in the CGM simulation.

We can show that the simulation of communications is simulated within the claimed slowdown in R1 and R2, even when the bandwidth of a superstep is different from h_{avg}. The details are omitted.

References

[1] P. Berenbrink, F. M. auf der Heide, and V. Stemann. Fault-tolerant shared memory simulations. In *Proc. 13-th Annual Symp. on Theoretical Aspects of Computer Science*, pages 181–192, 1996.

[2] J. F. Buss, P. C. Kanellakis, P. L. Ragde, and A. A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 10(1):45–86, 1996.

[3] B. S. Chlebus, A. Gambin, and P. Indyk. PRAM computations resilient to memory faults. In *Proc. of 2-nd Annual European Symp. on Algorithms*, volume 855 of *LNCS*, pages 401–412. Springer-Verlag, 1994.

[4] B. S. Chlebus, A. Gambin, and P. Indyk. Shared-memory simulations on faulty DMM. In *Proc. 23-rd International Coloquium on Automata, Languages, and Programming*, 1996.

[5] B. S. Chlebus, L. Gąsieniec, and A. Pelc. Fast deterministic simulation of computations on faulty parallel machines. In *Proc. 3-rd Annual European Symp. on Algorithms*, volume 979 of *LNCS*, pages 89–101. Springer-Verlag, 1195.

[6] D. E. Culler, R. M. Karp, D. A. Paterson, K. E. S. A. Sahay, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4-th, ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[7] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proc. of ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[8] F. Dehne and S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. *International Journal of Parallel Programming*, 1997.

[9] K. Diks and A. Pelc. Reliable computations on the faulty EREW PRAM. *Theoretical Computer Science*, to appear.

[10] L. Gąsieniec and P. Indyk. Efficient parallel computing with memory faults. In *Proc. Foundation of Computation Theory*, 1997.

[11] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications. Technical Report PRG-TR-13-96, Oxford University Computing Laboratory, 1996.

[12] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous algorithms. Technical Report TR-10-92, Center for Research in Computing Technology, 1992.

[13] M. T. Goodrich. Communication-efficient parallel sorting. In *Proc. of 28-rd Annual ACM Symp. on Theory of Computing*, pages 247–256, 1996.

[14] M. T. Goodrich. Randomized fully-scalable BSP techniques for multi-searching and convex hull construction. In *Proc. of 8-th ACM-SIAM Symp. on Discrete Algorithms*, 1997.

[15] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[16] P. Kanellakis, D. Michailidis, and A. A. Shvartsman. Controlling memory access concurrency in efficient fault-tolerant parallel algorithms. *Nordic Journal of Computing*, 2(2):146–180, 1995.

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

[18] R. M. Karp, A. Sahay, E.Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Proc. 5-th ACM Symp. on Parallel Algorithms and Architectures*, pages 142–153, 1993.

[19] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite executions for very fast dependable parallel computing. In *Proc. of 23-rd Annual ACM Symp. on Theory of Computing*, pages 381–390, 1991.

[20] Z. M. Kedem, K. V. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. 22-nd Annual ACM Symp. on Theory of Computing*, pages 138–148, 1990.

[21] S. C. Kontogiannis, G. E. Pantziou, and P. Spirakis. Efficient computations on fault-prone BSP machines. In *Proc. 9-th ACM Symp. on Parallel Algorithms and Architectures*, 1997.

[22] S. C. Kontogiannis, G. E. Pantziou, P. Spirakis, and M. Yung. Efficient computations on fault-prone bsp machines. In *Proc. 10-th ACM Symp. on Parallel Algorithms and Architectures*, 1998.

[23] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann Publishers, Inc., 1992.

[24] T. Leighton, B. Maggs, A. Ranade, and S. Rao. Randomized algorithms for routing and sorting in fixed-connection networks. *Journal of Algorithms*, 1991.

[25] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proc. 29-th Annual Symp. on Foundations of Computer Science*, pages 256–271, 1988.

[26] A. Ranade. How to emulate shared memory. In *Proc. 28-th Annual Symp. on Foundations of Computer Science*, pages 185–194, 1987.

[27] A. Ranade. *Fluent Parallel Computation*. PhD. Thesis, Yale University, New Haven, CT, 1988.

[28] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[29] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.

[30] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:102–111, 1990.

[31] L. G. Valiant. General purpose parallel architectures. *J. van Leeuwen, editor, Handbook of Theoretical Computer Science*, pages 943–972, 1990.

[32] L. G. Valiant and G. Brebner. Universal schemes for parallel communication. In *Proc. 13-th ACM Symp. on Theory of Computing*, pages 263–227, 1981.