Alcom-FT Technical Report Series
ALCOMFT-TR-01-169

in: Proc. Int. Conf. on Computational Science (ICCS) 2001, Vol. II, pp. 628–637

Parallel Bridging Models and Their Impact on Algorithm Design*

Friedhelm Meyer auf der Heide and Rolf Wanka

Dept. of Mathematics and Computer Science
and Heinz Nixdorf Institute
Paderborn University
33095 Paderborn, Germany
{fmadh,wanka}@upb.de

June 13, 2001

Abstract

The aim of this paper is to demonstrate the impact of features of parallel computation models on the design of efficient parallel algorithms. For this purpose, we start with considering Valiant's BSP model and design an optimal multisearch algorithm. For a realistic extension of this model which takes the critical blocksize into account, namely the BSP* model due to Bäumker, Dittrich, and Meyer auf der Heide, this algorithm is far from optimal. We show how the critical blocksize can be taken into account by presenting a modified multisearch algorithm which is optimal in the BSP* model. Similarly, we consider the D-BSP model due to de la Torre and Kruskal which extends BSP by introducing a way to measure locality of communication. Its influence on algorithm design is demonstrated by considering the broadcast problem. Finally, we explain how our Paderborn University BSP (PUB) Library incorporates such BSP extensions.

1 Introduction

The theory of efficient parallel algorithms is very successful in developing new algorithmic ideas and analytical techniques to design and analyze efficient parallel algorithms. The *P*arallel *R*andom *A*ccess *M*achine model (PRAM model) has proven to be very convenient for this purpose. On the other hand, the PRAM cost model (mis-)guides the algorithm designer to exploit a huge communication volume, and to use it in a fine-grained fashion. This happens because the PRAM cost model charges the same cost for computation and communication. In real parallel machines, however, communication is much more expensive than computation, and the cost for computation differs from machine to machine. Thus, it might happen that two algorithms for the same problem are incomparable in the sense that one is faster on machine A, the other is faster on machine B.

To overcome these problems, several proposals for so-called parallel bridging models have been developed: for example, the BSP model [16], the LogP model [6], the CGM model [7], and the QSM model [1].

*Partially supported by DFG SFB 376 "Massively Parallel Computation" and by the IST programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

A bridging model aims to meet the following goals: Its cost measure should guide the algorithm designer to develop efficient algorithms. It should be detailed enough to allow an accurate prediction of the algorithms' performance. It ought to provide an environment independent from a specific architecture and technology, yet reflecting the most important constraints of existing machines. This environment should also make it possible to write real portable programs that can be executed efficiently on various machines.

Valiant's BSP model [16] (*B*ulk *S*ynchronous model of *P*arallel computing) is intended to bridge the gap between software and hardware needs in parallel computing. In this model, a parallel computer has three parameters that govern the runtime of algorithms: The number of processors, the latency, and the gap.

The aim of this paper is to demonstrate the impact of features of the bridging model used on the design of efficient algorithms. For this purpose, in Section 2, after a detailed description of the (plain) BSP model, we present an efficient algorithm for the multisearch problem. Then, in Section 3, we explain that it is sometimes worthwhile to consider additional parameters of the parallel machine. Examples for such parameters are the *critical block size* and the *locality function* of the machine. The effect of the critical block size is demonstrated by presenting a multisearch algorithm that is efficient also if the critical block size is considered. The benefit of taking the locality into account is shown by presenting an efficient algorithm for the broadcast problem. In Section 4, we report on the implementation of the PUB Lib, the Paderborn BSP Library.

2 The Plain BSP Model

The PRAM model is one of the most widely used parallel computation models in theoretical computer science. It consists of a number of sequential computers that have access to a shared memory of unlimited size. In every time step of a PRAM's computation, a processor may read from or write into a shared memory location, or it can perform a single local step. So it charges one time unit both for an internal step, and for accessing the shared memory though these tasks seem to be quite different. The PRAM model is very comfortable for algorithm design because it abstracts from communication bottlenecks like bandwidth and latency. However, this often leads to the design of communication intensive algorithms that usually show bad performance if implemented on an actual parallel computer. In order to overcome this mismatch between model and actual machine, Valiant identified some abstract parameters of parallel machines that enable algorithm designer to charge different cost for the different tasks, without to be committed to a special parallel computer, but with a hopefully reliable prediction of the algorithms' performance. It is called the BSP model and discussed next.

2.1 Definition

A BSP machine is a parallel computer that consists of p processors, each processor having its local memory. The processors are interconnected via an interconnection mechanism (see Fig. 1(a)).

Algorithms on a BSP machine proceed in *supersteps* (see Fig. 1(b)). We describe what happens in a superstep from the point of view of processor P_i: When a new superstep t starts, all $\mu_{i,t}$ messages sent to P_i during the previous superstep are available at P_i. P_i performs some local computation, or work, that takes time $w_{i,t}$ and creates $\lambda_{i,t}$ new messages that are put into the interconnection mechanism, but cannot be received during the current superstep. Let $h_{i,t} = \max\{\mu_{i,t}, \lambda_{i,t}\}$. This superstep is finished by executing a barrier synchronization. Summarizing over all processors, we introduce the parameters $w_t = \max_i w_{i,t}$ and $h_t = \max_i h_{i,t}$. If an algorithm performs T supersteps, we use $W = \sum_{1\le t\le T} w_t$ and

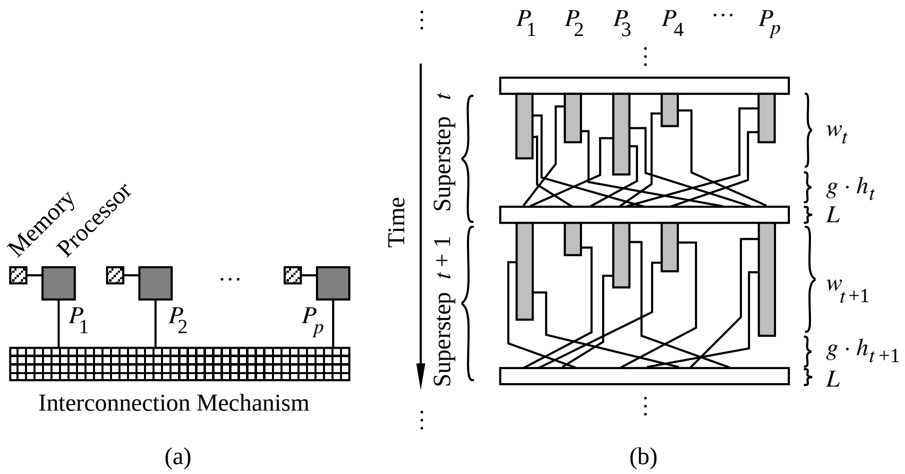

Figure 1: (a) A BSP computer and (b) the execution of a BSP algorithm.

$H = \sum_{1 \leq t \leq T} h_t$. W is called the local work of the algorithm, H its communication volume. Note that the size of the packets sent is not taken into account. In the course of analyzing a BSP algorithm, the task is to determine concrete values for T, H, and W.

The actual time that a BSP machine needs to execute the above BSP algorithm depends on the following machine parameters: L, the *latency*, and g, the *gap*, or *bandwidth inefficiency*. L is the maximum time that a message sent from a processor P_i needs to reach processor P_j, taken over all i and j. Likewise, L is also the time necessary for a single synchronization. Every processor can put a new packet into the interconnection mechanism after g units of time have been elapsed. Concrete values of g and L can be measured by experiments. For a different platforms catalogue, see [14]. The third machine parameter is p, the *number of processors*, or *the size of the machine*.

Hence, the runtime of the tth superstep is (at most) $w_t + g \cdot h_t + L$, and the overall runtime is (at most) $W + g \cdot H + L \cdot T$.

Note that we consider sums of times although it is often sufficient to only consider the maximum of the involved times when, e. g., pipelining can be used. This only results in a constant factor-deviation, whereas it simplifies many analyses considerably.

Let T_{seq} be the runtime of a best sequential algorithm known for a problem. Ideally, we are seeking BSP algorithms for this problem where $W = c \cdot T_{\text{seq}}/p$, $L \cdot T = o(T_{\text{seq}}/p)$, and $g \cdot H = o(T_{\text{seq}}/p)$, for a small constant $c \geq 1$. Such algorithms are called *c-optimal*. There are two popular ways to represent results of a BSP analysis. First, values of T, H, and W are given. For example, see Theorem 1 below. The other way is to state (with n denoting the input size) for which ranges of n/p, g, and L the algorithm is 1-optimal. Theorem 2 concluded from Theorem 1 is presented in this way.

2.2 An Example: The Multisearch Problem

As an instructive example, we outline an efficient BSP algorithm for the Multisearch Problem. Let U be a *universe* of objects, and $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ a partition of U into segments $\sigma_i \subseteq U$. The segments are *ordered*, in the sense that, for every $q \in U$ and segment σ_i, it can be determined (in constant time) whether $q \in \sigma_1 \cup \cdots \cup \sigma_{i-1}$, or $q \in \sigma_i$, or $q \in \sigma_{i+1} \cup \cdots \cup \sigma_m$. In the (m,n)-multisearch problem, the goal is to determine, for n objects (called *queries*) $q_1, \ldots, q_n$ from U, their respective segments. Also the queries are ordered, but we are not allowed to conclude from $q \in \sigma_i$ that $q' \in \sigma_j$, for any pair i, j.

Such problems arise in the context of algorithms in Computational Geometry (e. g., see Fig. 2).

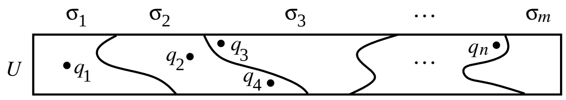

Figure 2: A multisearch instance from Computational Geometry. Note that q_3 and q_4 are in different relative orderings with respect to each other and their respective segments.

For simplicity, we assume that $m = d^c$ for some constant c and that the segments are given in the form of a complete d-ary search tree $\mathcal{T}$ of height $\log_d m + 1$. d will be determined later. Every leaf represents a segment, the leftmost leaf σ_1, the rightmost σ_m. Every inner node contains $d-1$ copies (called splitters) of the segments that split all its leaves into equally-sized intervals. A query q takes a path from the root (on level 0) of $\mathcal{T}$ to its respective leaf (on level $\log_d m$). In every inner node, it can be decided in time $O(\log d)$ by a binary search on the splitters to which child one has to go to resume the search.

In a seminal paper [13], Reif/Sen introduced a randomized PRAM algorithm that solves any $(O(n), n)$-multisearch instance on n processors in time $O(\log n)$, with high probability (w. h. p.), no shared memory cell being accessed at the same step more than once. A direct adaptation of this algorithm to the BSP model yields $T = W = H = O(\log n)$ which is far away from being optimal in the sense discussed above.

In the BSP setting, the parallel solution for the multisearch problem has to accomplish the following two tasks: (i) *Mapping*. In a preprocessing step, the nodes of $\mathcal{T}$ are mapped to the processors of the BSP machine. (ii) *Parallel multisearch*. After (i), the search for the segments of the n queries is performed in parallel.

The BSP algorithm works quite simple from a high-level point of view. The queries are distributed among the processors. Initially, they are all assigned to the root of $\mathcal{T}$, and they all must travel from the root to the correct leaf. In round t, it is determined for every query that is assigned to a node on level $t-1$ to which node on level t it has to go, so the number of rounds is $\log_d m$. As it turns out, the number of supersteps per round can be constant for a proper choice of d and a broad range for the BSP parameters.

The crucial task of this algorithm is the way of how the information to which node v of $\mathcal{T}$ which is stored in some processor P_i query q has to go meets q which is stored in some processor P_j. Should v be sent to P_j, or should q be sent to P_i? We shall see that answering this question individually for pairs (q, v) is the main step in the design of an efficient BSP algorithm.

Two kinds of hot spots can emerge that, unfortunately, are not addressed in a PRAM setting. The first kind can happen at a *processor*: There can be many nodes stored in a single processor that will be accessed in one round, either by receiving queries, or by sending their information away. There is a surprisingly simple way to tackle this problem. As for the mapping used in our plain BSP algorithm, we assume that the $\Theta(m)$ nodes of $\mathcal{T}$ are mapped randomly to the processors. The important observation for this mapping is: If $m \geq p \log p$, every processor manages $(1 + o(1)) \cdot m/p$ nodes, with high probability. We shall see in the next section that this mapping cannot avoid some problems if the so-called critical block size is considered as an additional parameter. However, for the time being, this simple way of mapping avoids hot spots at the processors. Of course we shall avoid sending requested information more than once. This is the aim of the solution of the problems arising from the second kind of hot spots.

Namely, the other kind of hot spots can emerge at *nodes* of $\mathcal{T}$. There can be nodes which lie on the paths of many queries from the nodes to the leaves. E. g., the root is on the path of every query, and, hence, a hot spot. The idea is to handle the case that many paths go through a nodes differently from the other case.

This distinction and separate handling is the impact of the BSP model on the design of an efficient BSP algorithm for the multisearch problem.

For a node v, let $J(v)$ be the set of queries that go through v. $J(v)$ is called the job of v. $J(v)$ is called a *large* job, if $|J(v)| > r$, otherwise, it is called a *small* job. For our purposes, it is sufficient to choose $r = (n/p)^{1-\varepsilon}$ for some ε, $0 < \varepsilon < 1$.

Let $v_1, \ldots, v_{d^t}$ be the nodes on level t of $\mathcal{T}$. Let the jobs be distributed among the processors as shown in Figure 3. For small jobs $J(v_i)$, $J(v_i)$ is sent to the processor that manages node v_i where the

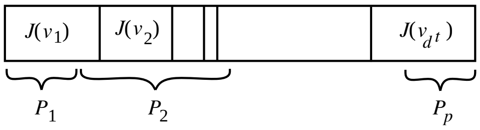

Figure 3: Distribution of the jobs.

nodes of the next level are computed by binary search. This can be done easily.

Large jobs $J(v_i)$ are handled differently because we want to avoid to send too many queries to one processor. Here the strategy is to first distribute $J(v_i)$ evenly on a group of consecutive processors such that at most n/p queries are on each of these processors. Then the group's first processor receives the management information, i. e., the splitters, from the processor that manages v_i and broadcasts it to the rest of the group. This routine can be implemented by using integer sorting and a segmented parallel prefix for which efficient BSP algorithms have to be used. The development of such algorithms is a very interesting work in its own right, but for our purposes it suffices to state the following theorem. Its proof can be found in [2, 3]. Note that a simple consideration shows that in the case of small jobs, it can be bad to send many 'large' nodes to the presumably many small jobs on a single processor.

Theorem 1 *Let $T_{\text{bin}}(x,y)$ denote the sequential worst case time for x binary searches on a sequence of y segments. Let $c \geq 1$, $1 < k < p$ and $d = o((n/p)^{1/c})$.*

The multisearch algorithm presented above performs n searches on a d-ary tree of depth δ in time $\delta \cdot (W + g \cdot H + T \cdot L)$, w.h.p., with

$$\begin{aligned} W &= (1+o(1)) \cdot T_{\text{bin}}(n/p,d) + O(k \cdot ((n/p)^{1/c} + c + \log_k p)) \enspace , \\ H &= O(c \cdot (n/p)^{1/c} + k \cdot \log_k p) \enspace , \\ T &= O(c + \log_k p) \enspace . \end{aligned}$$

With $T_{\text{bin}}(x,y) = O(x \log y)$ and, therefore, $T_{\text{seq}} = O(\delta \cdot n/p \cdot \log d)$ one can compute for which ranges of the machine parameters p, g and L the algorithm is 1-optimal.

Theorem 2 *Let $c \geq 1$ be an arbitrary constant and $d = (n/p)^{1/c}$. Then the algorithm solves the multisearch problem with n queries on a d-ary tree with depth $\delta = \log_d p$ in runtime $\delta \cdot (1 + o(1)) \cdot T_{\text{bin}}(n/p,d)$ for the following parameter constellations:*

- *$n/p = \Omega((\log p)^c)$, $g = o(\log(n/p))$, and $L = o(n \log(n/p)/(p \log p))$.*
- *$n/p = \Omega(p^\varepsilon)$, for any $\varepsilon > 0$, $g = o(\log(n/p))$, and $L = (n/p) \cdot \log(n/p)$.*

3 Extensions of BSP

As mentioned in the introduction, the runtime computed with the BSP parameters should allow an accurate prediction of the algorithms' performance. However, quite often it can be observed that the prediction and the actual performance of an implementation deviate considerably despite a tight analysis. That means that at least in such cases, there are properties of existing machines that influence the performance heavily, but that are not visible in the BSP approach and therefore missing when algorithms are designed. In this section, two such properties are identified and incorporated into the BSP model. For both variations of the BSP model, we present algorithms that show the impact of the changes in the model on the algorithm design.

3.1 BSP*: Critical Block Size and the Multisearch Problem Revisited

Observations of existing interconnection mechanisms of real machines show that it is sometimes better to send large packets that contain many single information units than sending all these units separately. But – as mentioned above – the size of messages is not taken into account in the BSP model. In order to model this aspect, the *critical block size* B has been introduced as additional parameter [2, 8] resulting in the BSP* model. B is the minimum number of information units a message must have in order to fully exploit the bandwidth of the communication mechanism. This also leads to a modified definition of the gap: Every processor can put a new packet *of size* B into the communication mechanism after g^* units of time have been elapsed.

Now for superstep t, we count how many messages h_t are sent or received by a single processor, and how many information units s_t are parceled in these messages. For the runtime of the superstep, we charge $w_t + g^* \cdot (s_t + h_t \cdot B) + L$. Note that if h_t messages containing altogether $s_t = h_t \cdot B$ information units are sent, the runtime is $w_t + 2g^* \cdot s_t + L$. If h_t messages containing only $s_t = h_t$ information units are sent, the runtime is $w_t + g^* \cdot h_t \cdot (B+1) + L$. That means that short messages are treated as if they were of size B. The number T of supersteps and the local work W of an BSP* algorithm are identical to those in the BSP model. Now, $H = \sum_t h_t$ is the *number of message start-ups*, and the *communication volume* is $S = \sum_t s_t$. The total runtime of a BSP* algorithm is $W + g^*(S + B \cdot H) + T \cdot L$.

Let H_{BSP} denote the number of messages of a BSP algorithm. In a bad case, it can be that every message contains only one information unit. That means the BSP* runtime is $W + g^*(1+B) \cdot H_{\text{BSP}} + T \cdot L$. In this case, we are seeking for a new (or modified) algorithm for the BSP* machine that communicates (about) the same amount of information, but parceled into at most $H = H_{\text{BSP}}/B$ messages.

Indeed, the bad case mentioned above can occur in the BSP multisearch algorithm presented in Subsection 2.2. For example, suppose a very large search tree $\mathcal{T}$ and few queries that all belong to different leaves of $\mathcal{T}$. If we run our BSP algorithm with a random mapping of the nodes of $\mathcal{T}$, there will be a level (i. e., round) t_0, after that all jobs have size 1 and, due to the good distribution property of a random mapping, all jobs have to be sent from now on to different processors, w. h. p. That means that we cannot create large messages to exploit the bandwidth of the communication mechanism, and that almost all messages that have size 1 are charged with B. In fact, this computed runtime comes in total much closer to the actual performance of this algorithm on many real machines than the plain BSP runtime.

From the discussion above, it follows that the random mapping of the nodes of $\mathcal{T}$ causes the problems. In the following, we shall describe a different mapping that enables the algorithm also to parcel large messages after some critical level t_0 of $\mathcal{T}$ as described above has been reached. This mapping is the so-called z-mapping.

It is the impact of considering the critical block size in the BSP model on the design of an efficient algorithm for the multisearch problem.*

Note that also adapted BSP* algorithms for integer sorting, segmented parallel prefix and broadcast have to be designed and applied as subroutines.

Let $z \leq p$. The z-mapping of the nodes of a d-ary tree $\mathcal{T}$ with $\delta + 1$ levels works as follows, with t going from 0 through δ iteratively: On level t, $t \leq \log_d p$, there are at most p nodes which are randomly mapped to different processors. On level t, $t > \log_d p$, there are more than p nodes. For every processor P, let $R(P)$ be the subset of nodes in level $t-1$ that has been mapped to P. All children of the nodes of $R(P)$ are distributed randomly among z randomly chosen processors.

In the random mapping, the children of $R(P)$ can be spread evenly among *all* processors, whereas in the z-mapping they are, so to speak, clustered to only z processors where they are spread evenly.

Now we observe how the multisearch algorithm has to behave in the case of small jobs if the z-mapping is applied. Large jobs are treated as before.

After the partition of jobs of level t-nodes has been completed, every small job $J(v)$ is stored in processor P that also stores v. P splits (by binary search) $J(v)$ into the jobs for the children nodes on level $t+1$. The z-mapping ensures that these children are scattered only on at most z processors. Even if P has many small jobs, all children nodes are distributed among at most z processors. So P can create z large messages.

Of course, this description can only put some intuition across the success of this mapping. A detailed analysis of this algorithm [2, 3] leads to the following theorem that states when this algorithm is an optimal parallelization.

Theorem 3 *Let $c \geq 1$ be an arbitrary constant and choose $d = (n/p)^{1/c}$ and let $z = \omega(d \log p)$. Let $\mathcal{T}$ be a d-ary tree with depth δ that has been mapped onto the processors by using the z-mapping.*

W. h. p., the runtime of the above algorithm is $\delta \cdot (1 + o(1)) \cdot T_{\text{bin}}(n/p, d)$, i. e. 1-optimal, for the following parameter constellations:

- $n/p = \omega((\log)^c)$, $g^* = o(\log(n/p))$, $B = o((n/p)^{1-1/c})$, *and* $L = o((n \log(n/p)/(p \log p))$.
- $n/p = \Omega(p^\varepsilon)$, *for an arbitrary constant* $\varepsilon > 0$, $g^* = o(\log(n/p))$, $B = o((n/p)^{1-1/c})$, *and* $L = (n/p)\log(n/p)$.

3.2 D-BSP: Locality and the Broadcast Problem

Many parallel algorithms work in a recursive way, i. e., the parallel machine is split into two or more independent parts, or sets of processors, where similar subproblems of smaller size, have to be solved. It is often possible to partition a real parallel machine into independent parts such that we can consider the latency and the gap as functions $L(k)$ and $g(k)$, resp., of the size k of a set of processors because it can make sense that, e. g., the latency on a small machine is much smaller than on a huge machine. In the *decomposable* BSP machine model (D-BSP) introduced by de la Torre and Kruskal [15], in a step the machine can be partitioned into two equally-sized submachines. This partitioning process may be repeated on some submachines (i. e., submachines of different sizes are possible simultaneously). In the D-BSP model, the runtime of a superstep *on a single submachine* is $w_t + h_t \cdot g(p_t) + L(p_t)$, with p_t denoting the size of this submachine. The runtime of a superstep is the maximum runtime taken over the submachines.

In the following, we present an algorithm for the broadcast problem that can easily be implemented on the D-BSP machine. In the broadcast problem, processor P_1 holds an item α that has to be sent to the remaining $p-1$ processors.

An obvious plain BSP algorithm for this problem works in $\log p$ supersteps. In superstep t, every processor P_i, $1 \leq i \leq 2^{t-1}$ sends α to processor P_{i+2^t} (if this processor exists). The BSP parameters are $T = W = H = O(\log p)$ so the BSP runtime is $O(\log p + g \cdot \log p + L \cdot \log p)$.

For example, suppose the interconnection mechanism to be a hypercube with Gray code numbering, guaranteeing $L(2^k) = O(k)$, and suppose $g(2^k) = O(1)$. Then the runtime of the plain BSP algorithm implemented directly on this D-BSP machine is $\Theta((\log p)^2)$.

The following recursive broadcast algorithm dubbed $\textsc{Root}(p)$ has been introduced by Juurlink *et al.* [12] (also see [14]): If there are more than two processors, $\textsc{Root}(\sqrt{p})$ is executed on $P_1, \ldots, P_{\sqrt{p}}$ (appropriate rounding provided). Then every processor P_i, $i \in \{1, \ldots, \sqrt{p}\}$, sends α to $P_{i \cdot \sqrt{p}}$. Now, the machine is partitioned in $\sqrt{p}$ groups of size $\sqrt{p}$ each, where finally $\textsc{Root}(\sqrt{p})$ is executed.

It is a nice exercise to write a D-BSP program that implements algorithm $\textsc{Root}(p)$ and to prove the following theorem.

The (recursive) partition of the problem into subproblems that have to be solved on 'compact' submachines is the impact of the locality function on the algorithm design.

Theorem 4 *On the D-BSP, algorithm* $\textsc{Root}(p)$ *has a runtime of*

$$O\left(\sum_{i=0}^{\log\log p} 2^i \cdot \left(L(p^{1/2^i}) + g(p^{1/2^i})\right)\right) .$$

For our hypercube example, the runtime of $\textsc{Root}(p)$ is $O(\log p \log\log p)$.

Correspondingly to Theorem 4, Juurlink *et al.* prove a lower bound [12] that matches this upper bound for large ranges for $L(k)$ and $g(k)$.

Theorem 5 *Any algorithm for the broadcast problem on a p-processor D-BSP machine with latency* $L(k)$ *and gap* $g(k)$ *with* $\log k / \log\log k \leq L(k) \leq (\log k)^2$ *and*
$\log k / \log\log k \leq g(k) \leq (\log k)^2$ *for all k,* $1 \leq k \leq p$*, has runtime*

$$\Omega\left(\sum_{i=0}^{\log\log p} 2^i \cdot \left(L(p^{1/2^i}) + g(p^{1/2^i})\right)\right) .$$

4 The PUB Lib

In the previous sections, we showed how the BSP model and its extensions influence the design of efficient parallel algorithms. The other important aim of a bridging model like, in our case, BSP is to achieve portability of programs. That means there should be an environment on parallel machines that supports the coding of BSP algorithms independent from the underlying machine and that executes the code efficiently.

Such BSP environments are the Oxford BSPlib [10], the Green library [9], and the PUB Lib (Paderborn University BSP Library) [5].

The PUB Lib is a C Library to support the development and implementation of parallel algorithm designed for the BSP model. It provides the use of block-wise communication as suggested by the BSP* model, and it provides the use of locality as suggested in the D-BSP model by allowing to dynamically partition the machine into independent submachines.

It has a number of additional features:

- Collective communication operations like, e. g., broadcast and parallel prefix are provided and implemented in an architecture independent way. Furthermore, as they are non-synchronizing, they have very good runtimes.
- By providing partitioning and subset synchronization operations it is possible for users of the PUB Lib to implement and analyze algorithms for locality models as discussed in the previous section.
- Oblivious synchronization is provided.
- Virtual processors are incorporated into the PUB Lib. These processors can be used for solving problems of sizes that do not fit into main memory efficiently on a sequential machine. Furthermore, these virtual processors serve as a parallel machine simulator for easy debugging on standard sequential programming environments.

The PUB Lib has been successfully installed on the Cray T3E, Parsytec CC, GCel, GCPP, IBM SP/2, and workstation clusters, and on various single CPU systems.

All information for downloading and using the PUB Lib can be found on the project's webpage http://www.upb.de/~pub/.

Currently, the implementation of the migration of virtual processors from heavily loaded CPUs to less loaded CPUs is in progress.

Acknowledgments

The effort on BSP, BSP*, and on designing and analyzing algorithms for them and on implementing the PUB Lib in Paderborn is the work of many individuals: Armin Bäumker, Olaf Bonorden, Wolfgang Dittrich, Silvia Götz, Nicolas Hüppelshäuser, Ben Juurlink, Ingo von Otte, and Ingo Rieping who are certainly also authors of this paper.

References

[1] M. Adler, P.B. Gibbons, Y. Matias, V. Ramachandran. Modeling parallel bandwidth: local versus global restrictions. *Algorithmica* 24 (1999) 381–404.

[2] A. Bäumker. *Communication Efficient Parallel Searching*. Ph.D. Thesis, Paderborn University, 1997.

[3] A. Bäumker, W. Dittrich, F. Meyer auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. *TCS* 203 (1998) 175–203.

[4] A. Bäumker, F. Meyer auf der Heide. Communication efficient parallel searching. In: *Proc. 4th Symp. on Solving Irregularly Structured Problems in Parallel (IRREGULAR)*, 1997, pp. 233–254.

[5] O. Bonorden, B. Juurlink, I. von Otte, I. Rieping. The Paderborn University BSP (PUB) Library — Design, Implementation and Performance. In: *Proc. 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, 1999, pp. 99–104.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian R., T. von Eicken. LogP: A practical model of parallel computation. *C.ACM* 39(11) (1996) 78–85.

[7] F. Dehne, A. Fabri, A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *Int. J. Computational Geometry & Applications* 6 (1996) 379–400.

[8] W. Dittrich. *Communication and I/O Efficient Parallel Data Structures*. Ph.D. Thesis, Paderborn University, 1997.

[9] M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas. Portable and efficient parallel computing using the BSP model. *IEEE Transactions on Computers* 48 (1999) 670–689.

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

[11] B.H.H. Juurlink. *Computational Models for Parallel Computers*. Ph.D. Thesis, Leiden University, 1997.

[12] B.H.H. Juurlink, P. Kolman, F. Meyer auf der Heide, I. Rieping. Optimal broadcast on parallel locality models. In: *Proc. 7th Coll. on Structural Information and Communication Complexity (SIROCCO)*, 2000, pp. 211–226.

[13] J.H. Reif, S. Sen. Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications. *SIAM J. Computing* 23 (1994) 633–651.

[14] I. Rieping. *Communication in Parallel Systems – Models, Algorithms and Implementations*. Ph.D. Thesis, Paderborn University, 2000.

[15] P. de la Torre, C. P. Kruskal. Submachine locality in the bulk synchronous setting. In: *Proc. 2nd European Conference on Parallel Processing (Euro-Par)*, 1996, 352–358.

[16] L. Valiant. A bridging model for parallel computation. *C.ACM* 33(8) (1990) 103–111.