



## The Paderborn University BSP (PUB) library <sup>☆</sup>

Olaf Bonorden <sup>a</sup>, Ben Juurlink <sup>b,\*</sup>, Ingo von Otte <sup>a</sup>,  
Ingo Rieping <sup>a</sup>

<sup>a</sup> *Heinz Nixdorf Institute and Department of Computer Science, Paderborn University, Germany*

<sup>b</sup> *Computer Engineering Laboratory, Electrical Engineering Department,  
Delft University of Technology, The Netherlands*

Received 12 February 2001; received in revised form 25 June 2002; accepted 5 October 2002

---

### Abstract

The Paderborn University BSP (PUB) library is a C communication library based on the BSP model. The basic library supports buffered as well as unbuffered non-blocking communication between any pair of processors and a mechanism for synchronizing the processors in a barrier style. In addition, PUB provides non-blocking collective communication operations on arbitrary subsets of processors, the ability to partition the processors into independent groups that execute asynchronously from each other, and a zero-cost synchronization mechanism. Furthermore, some techniques used in the implementation of the PUB library deviate significantly from the techniques used in other BSP libraries.

© 2002 Elsevier Science B.V. All rights reserved.

*Keywords:* Parallel communication library; Bulk synchronous parallel model; Nested parallelism; Oblivious synchronization; Collective communication

---

### 1. Introduction

Most message-passing libraries are based on pairwise sends and receives. This means that for each send operation, a matching receive has to be issued on the destination processor. Widely available message-passing libraries like PVM [9] and MPI

---

<sup>☆</sup> A preliminary version of this paper appeared in IPPS/SPDP'99. This research was supported in part by DFG SFB 376 "Massively Parallel Computation" and by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

\*Corresponding author.

*E-mail addresses:* [bono@uni-paderborn.de](mailto:bono@uni-paderborn.de) (O. Bonorden), [benj@ce.et.tudelft.nl](mailto:benj@ce.et.tudelft.nl) (B. Juurlink), [ivo@uni-paderborn.de](mailto:ivo@uni-paderborn.de) (I. von Otte), [inri@uni-paderborn.de](mailto:inri@uni-paderborn.de) (I. Rieping).

[22] provide the user with this kind of programming model. This approach, however, is very error prone because deadlocks can be easily introduced if a message is never accepted. Furthermore, it is difficult to determine the correctness and time complexity of programs implemented using pairwise sends and receives.

An alternative parallel programming model is provided by the bulk synchronous parallel (BSP) model [31]. In this model, a parallel computation consists of a sequence of phases called *supersteps*. In every superstep, each processor can perform operations on data present in its local memory at the beginning of the superstep, send some messages, and (implicitly) receive some messages. Supersteps are separated by a barrier synchronization. If a new superstep begins, it is assured that all messages sent in the previous superstep have reached their destinations. This approach is less prone to deadlocks because there are no explicit receives. Instead, a barrier synchronization signifies the end of all communication operations. In other words, communication is *one-sided*. Moreover, the BSP model provides a simple cost model that makes it relatively easy to determine the cost of executing a parallel program. For more details on and motivation for the BSP model, the reader is referred to [21,31].

In this paper we present the Paderborn University BSP (PUB) library. It is based on the BSP model but supports a number of additional features. First, in addition to providing basic BSP functionality (i.e., point-to-point communication and barrier synchronization), it provides a rich set of collective communication operations like broadcast, reduce and scan. These primitives are non-blocking and can be executed by an arbitrary subset of processors. The Oxford BSP library [24] and the Green BSP library [14] do not provide such routines. The Oxford BSP toolset implementation of BSPlib [15] provides a collective communication library but the routines in this library are blocking, involve all processors, and are implemented on top of the core operations. Because these operations occur frequently and often determine the overall efficiency of a parallel application, the implementation of these operations in the PUB library exploits architectural features not visible to the user of a BSP library.

Another distinguishing feature is the possibility to dynamically partition the processors into independent subsets. After a partition operation, each subsystem acts as an autonomous BSP computer which means that subsequent barrier synchronizations involve only the processors in the subsystem. Thus, the PUB library supports nested parallelism and subset synchronization. This feature is especially useful when completely different subalgorithms have to be performed independently in parallel, as is needed, e.g., in the parallelization of an ocean simulation program [28]. Without the possibility to create subsystems, the algorithms have to be interleaved, which complicates the implementation. Furthermore, the implementation might be inefficient if the algorithms have different synchronization requirements.

PUB also supports a zero-cost synchronization mechanism. This mechanism can be used if it is known in advance how many messages each processor is due to receive, and we call it *oblivious synchronization*. It is motivated by the observation that a standard barrier synchronization can be very expensive on many parallel architectures and, moreover, its cost increases with the number of processors.

Another useful feature is the concept of *BSP objects*. These objects serve three purposes. First, they are used to distinguish the different processor groups that exist after a partition operation. Second, they are used for modularity and safety purposes. Third, they can be used to ensure that messages sent in different threads do not interfere with each other and that a barrier synchronization executed in one thread does not suspend the other threads running on the same processor.

Currently, the PUB library is available on the CRAY T3E, the IBM SP2, a PC cluster connected by a Fast Ethernet switch, the Intel Paragon, several parallel systems from Parsytec, and networks of workstations. An implementation on top of MPI is available too, so that PUB can be ported directly to all architectures on which MPI is available. There is also a simulator so that PUB programs can be developed and tested on a PC or workstation.

### 1.1. Related work

The Oxford BSP library [24] was the first BSP library. Basically, it contains functions for delimiting supersteps and so-called *direct remote memory access* (DRMA) operations. These DRMA operations transfer data to and from the local memory of another processor, similar to the primitives provided by the Cray T3D/T3E SHMEM library [3]. However, in the Oxford BSP library, only static variables can be accessed remotely.

The Green BSP library [14] does not provide DRMA operations but only one *bulk synchronous message passing* (BSMP) operation that sends a fixed-size packet to a specified destination processor. In contrast to traditional message-passing operations, there is no receive operation, but instead the message is stored in a queue on the destination processor, from which it can be extracted after the next barrier synchronization. This type of communication is similar to depositing a message in a mailbox. It is more convenient than DRMA when it is unknown a priori where the data must be stored on the destination processor.

BSPlib [15] provides DRMA as well as BSMP operations. Furthermore, the restriction that only static variables can be accessed remotely is eliminated by requiring that variables are registered before they are used in a DRMA operation. PUB offers the same functionality as BSPlib, but in addition provides several other primitives like non-blocking collective communication routines, partition operations, and other forms of synchronization besides standard barrier synchronization. In fact, PUB can be configured to accept BSPlib programs.

In addition to two-sided message-passing primitives, MPI-2 [23] also supports one-sided DRMA operations. MPI appears to be complex compared to the BSP model but could also be used to program applications in the BSP programming style.

Several authors (e.g., [2,8]) also recognized that a standard barrier synchronization can be very expensive and, therefore, modified the BSP model to include oblivious synchronization. These authors showed that many applications can employ this type of synchronization. Similar concepts can be found in [20] and [6]. A model for BSP with oblivious synchronization, motivated by the PUB library, was described in [13].

NestStep [19] and LLC [27] also support nesting of supersteps, i.e., partition operations. In [12,26,29], extensions of the BSP cost model are proposed that can be used to analyze algorithms that exploit nested parallelism.

## 1.2. Organization

In order to describe the PUB library, we have chosen a presentation that interweaves examples with descriptions of library functions and performance results. Section 2 describes most of PUB's bulk synchronous message-passing routines using parallel binary search as an example. Partition operations are also illustrated in this section. Section 3 describes a simple sorting algorithm and shows how it can be implemented using PUB's collective communication routines. Section 4 explains some other features of the PUB library, provides some implementation details, and presents experimental data supporting the claim that it is beneficial to implement frequently occurring communication operations directly on the architecture rather than on top of the core BSP operations. Concluding remarks are given in Section 5.

## 2. Example 1: Binary search

In this section some of the programming primitives provided by the PUB library are illustrated using parallel binary search as an example. First, we briefly describe the data structure and present an initial parallel program. After that, a sequence of optimizations is performed on the program that illustrate some of PUB's features.

### 2.1. Parallel binary search

In parallel binary search,  $M = mp$  search keys called *queries* have to be located in a search structure of size  $N = np$ , where  $p$  is the number of processors. We use a search structure that resembles a search-butterfly [1]. In this structure the root of the search tree is replicated  $p$  times (once on every processor) and each level below the root is replicated half as many times as the level above. Furthermore, the outputs of the butterfly are the roots of balanced binary search trees of size  $n$ . Fig. 1 depicts a search-butterfly with  $n = m = p = 4$  and illustrates how it is distributed across the processors.

To locate the queries in a search-butterfly, every query first has to be routed to the processor containing the correct subtree. This is done by sending them either to the left or right half at every level of the butterfly, depending on whether the query is smaller than or equal to the current node's key or not, as illustrated in Fig. 1 for processor 0. A processor  $p$  is said to belong to the left (resp. right) half of the butterfly in dimension  $d$ , if the  $d$ th least significant bit in the binary representation of  $p$  is a 0 (resp. 1). Of course, this simple approach causes load imbalance if all queries have to go to the same subtree, but for simplicity, we assume that the destinations of the queries are distributed evenly.

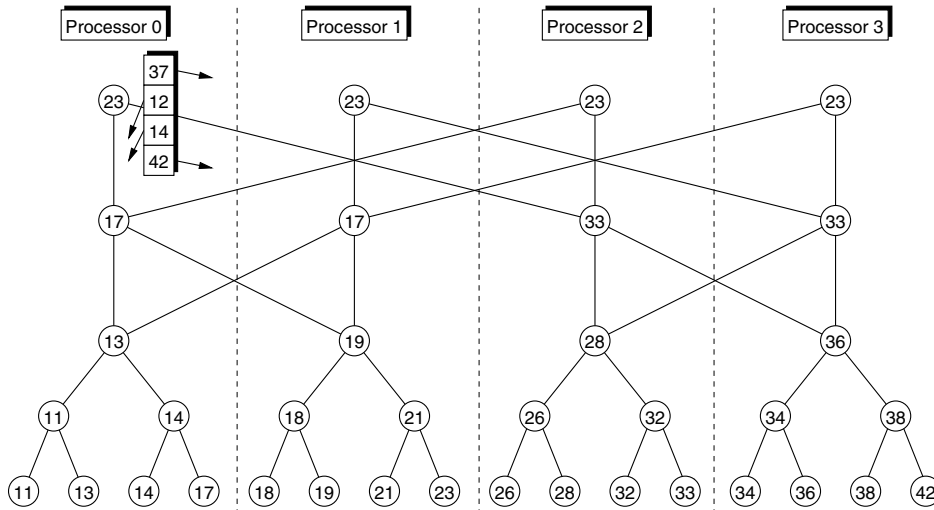


Fig. 1. A search-butterfly with  $n = m = p = 4$ .

## 2.2. An initial solution

Program 1 shows how parallel binary search can be implemented using PUB. For brevity, the declaration and initialization of the variables have been omitted. The queries are stored in the array `query`, and `gkey` is an array of size  $\log p$  containing the first  $\log p$  replicated levels of the search tree, in reverse order. For the search-butterfly depicted in Fig. 1, `gkey[2] = {17, 23}` for processors 0 and 1, and `gkey[2] = {33, 23}` for processors 2 and 3.

PUB follows the single program multiple data programming paradigm. Each processor has a unique ID in the range 0 to  $p - 1$ , which can be determined by calling `bsp_pid`. In Program 1 it is assumed that this ID has already been stored in the variable `mypid`.

The code in lines 3–8 routes each query to the processor containing the correct subtree. The macro's `inLeftHalf(dim, mypid)` and `inRightHalf(dim, mypid)` return true if processor `mypid` belongs to the left or right half of the butterfly in dimension `dim`, respectively, and the macro `Opposite(dim, mypid)` returns the ID of the processor opposite to processor `mypid` in dimension `dim`. The call `bsp_send(bsp, dest, buf, size)` in line 6 sends a message of `size` bytes to the processor with ID `dest`. This is a buffered message-passing primitive. The array locations that have been sent can, therefore, be reused immediately. The first argument of `bsp_send` is a pointer to an object of type `tbsp`, which must be passed to most PUB functions. It is used, for example, to distinguish different groups of processors. More details on the use of these objects are provided in Section 4.

**Program 1.** Initial parallel binary search program

```

1 void bin_search(int dim, int m)
2 {
3   for (i = new_m = 0; i < m; i++)
4     if ((query[i] <= gkey[dim] && inRightHalf(dim, mypid))
5         || (query[i] > gkey[dim] && inLeftHalf(dim, mypid)))
6       bsp_send(bsp, Opposite(dim, mypid), & query[i],
7               sizeof(int));
8     else
9       query[new_m++] = query[i];
10    bsp_sync(bsp);
11  for (i = 0; i < bsp_nmsgs(bsp); i++) {
12    msg = bsp_getmsg(bsp, i);
13    query[new_m++] = * (int *) bspmsg_data(msg);
14  }
15  if (dim == 0) local_search(new_m, query, n, key);
16  else          bin_search(dim-1, new_m);
17 }

```

The call to `bsp_sync` in line 9 performs a barrier synchronization. When this function returns, all messages sent in the previous superstep are available in a buffer on the destination processor. The number of messages in this buffer can be determined by calling `bsp_nmsgs`, and messages can be fetched from the buffer by calling `bsp_getmsg(bsp, i)`. This function returns a message handle `msg` to the `i`th message in the buffer, but does not remove it from the buffer (it exists until the next synchronization point). In other BSP programming libraries, messages can only be removed from the front of the buffer (i.e., the buffer is essentially a queue). We decided to provide random access into the message buffer, because first, this is more flexible, and second, messages do not always need to be copied. Sometimes, the computation can work directly on the message data. In our experience, frequent message copying contributes significantly to the communication overhead, especially on high bandwidth parallel systems where communication is (almost) as fast as a local memory access.

The message handle `msg` can be used to obtain the data part of the message (by calling `bspmsg_data(msg)`), as well as to determine the size of the message (`bspmsg_size(msg)`) and the ID of the source (`bspmsg_src(msg)`). As these functions operate directly on messages, no BSP object has to be supplied, which is why their names have the prefix `bspmsg` instead of `bsp`. To complete the description of Program 1, in lines 15–16, the search is either finished locally or binary search is called recursively.

The communication in Program 1 is rather fine grain. In order to amortize the startup cost of a message transmission, PUB offers the possibility to automatically combine small packets into larger ones, similar to many implementations of BSPLib [30]. In the following, the version of Program 1 that does not perform packet com-

binning will be referred to as Program 1a, and the version that does will be referred to as Program 1b.

The curves labeled (1a) and (1b) in Figs. 2–4 depict the performance achieved by Program 1a and Program 1b on a Parsytec CC, a PC cluster, and a Cray T3E, respectively. Information about these platforms is provided in Table 1. The *x*-axis

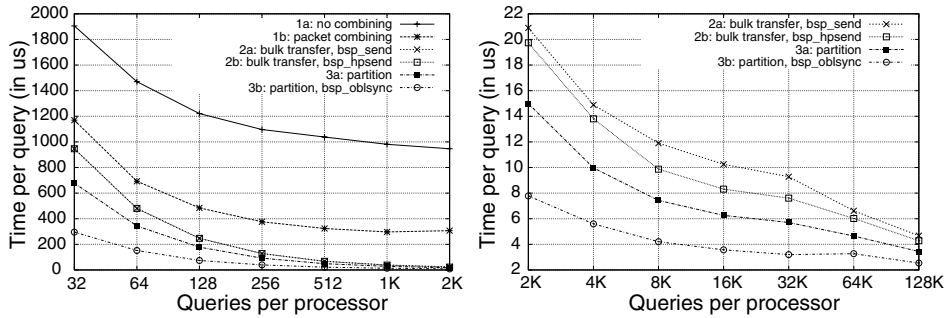


Fig. 2. Time per query per processor for several parallel binary search versions on the Parsytec CC.

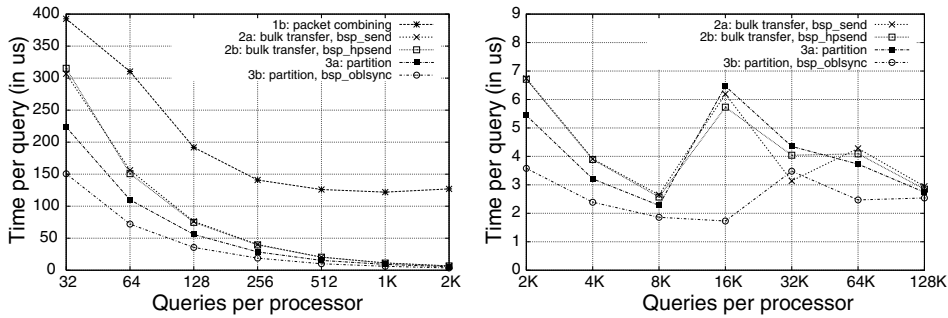


Fig. 3. Time per query for several parallel binary search versions on the PC cluster.

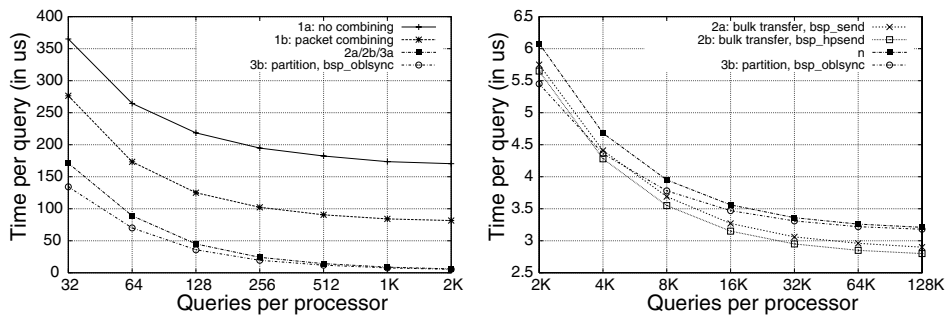


Fig. 4. Time per query for several parallel binary search version on the Cray T3E.

Table 1  
Parameters of the experimental platforms

Machine	$p$	Processor	Frequency (MHz)	Interconnect	Bandwidth	Operating system	Communication layer
PC cluster	16	Intel Pentium II (2)	450	Cisco Catalyst 5509 Fast Ethernet switch	100 Mb/s, 3.6 Gb/s	Linux	TCP/IP
Parsytec CC	32	Power PC 604	133	Fat mesh of clos	1 Gb/s	Parix	Parix
Cray T3E	64	DEC Alpha EV5	300	3D Torus	500 Mb/s, 1200 Mb/s	UNICOS	SHMEM

The nodes of the PC cluster consist of two Intel Pentium II processors and are connected by 100 Mb/s Fast Ethernet interfaces to a switch with an aggregate bandwidth of 3.6 Gb/s. The bandwidth of the external data bus of each T3E processor is 1200 Mb/s and the network bandwidth is 500 Mb/s. The last column depicts the communication layer on top of which PUB is implemented.

shows the number of queries per processor  $m$ , and the  $y$ -axis shows the time per query per processor ( $T/m$ ) needed for routing the queries to the right subtree. The running times do not include the time taken by the local search, and are maximized across the processors involved and averaged over many runs. The figures on the left depict the running times when the number of queries per processor is less than 2K, the figures on the right show the running times when the number of queries per processor is larger than 2K.

On all architectures considered packet combining improves performance significantly. For example, on the CC (Fig. 2), when the number of queries per processor is 2K, Program 1a requires 946  $\mu$ s per query, and Program 1b requires only 306  $\mu$ s per query. On the PC cluster (Fig. 3), the running times of Program 1a fall outside the displayed range, since it always requires 4.4 as much time as the program variant with packet combining. On the Cray T3E (Fig. 4), packet combining also improves performance significantly: when there are 2K queries per processor, Program 1b is faster than Program 1a by a factor of 2.1. However, in all cases, the figures on the right do not display the running times of the two program variants discussed so far. This is because the program variants that will be discussed next are significantly faster.

### 2.3. Bulk transfers

Although automatic packet combining improves performance significantly, sending many small packets still has a negative impact on performance for the following reasons [16]. First, several bytes of bookkeeping information have to be sent with every packet. If bulk transfers are used, a larger part of the packet can be used for user data. Second, on high-bandwidth systems frequent message buffering can contribute significantly to the communication overhead, because the time needed

to write a message into the output buffer not only includes the time needed to copy the message but also the time required to advance pointers, generate a message header, etc. Furthermore, our measurements [17] show that on the Cray T3E writing to a remote memory location is as fast as writing to a local memory location and that reading from a remote memory location is actually faster than a local memcpy. On such platforms the cost of copying is not negligible.

In order to be able to use bulk transfer in our binary search example, the data must be packed explicitly, as shown in Program 2. An auxiliary array `temp` is used, in which the queries that have to be routed to the other half of the butterfly are written. We also replaced the call to `bsp_send` by a call to `bsp_hpsend`. This function has the same synopsis as `bsp_send` but is unbuffered, thereby avoiding message buffering. This implies, however, that it is not safe to alter the data during the super-step.

**Program 2.** Parallel binary search using bulk transfers

```

1 void bin_search(int dim, int m)
2 {
3   for (i = new_m = other = 0; i < m; i++)
4     if ((query[i] <= gkey[dim] && inRightHalf(dim, mypid))
5         || (query[i] > gkey[dim] && inLeftHalf(dim, mypid)))
6       temp[other++] = query[i];
7   else
8     query[new_m++] = query[i];
9
10  bsp_hpsend(bsp, Opposite(dim, mypid), temp,
11            other* sizeof(int));
12  bsp_sync(bsp);
13  msg = bsp_getmsg(bsp, 0);
14  memcpy(&query[new_m], bspmsg_data(msg),
15        bspmsg_size(msg));
16  new_m += bspmsg_size(msg) / sizeof(int);
17  if (dim == 0) local_search(new_m, query, n, key);
18  else bin_search(dim-1, new_m);
19 }
```

The version of Program 2 that uses `bsp_send` will be referred to as Program 2a and the version that uses `bsp_hpsend` will be referred to as Program 2b. The curves labeled (2a) and (2b) in Figs. 2–4 depict the performance achieved by Program 2a and 2b, respectively. It can be observed that on all three architectures, employing bulk transfers instead of relying on automatic packet combining yields much better performance. For example, when there are 2K keys per processor, Program 2a is faster than Program 1b by a factor of 13 on the CC, by a factor of 19 on the PC cluster, and by a factor of 14 on the T3E. Moreover, the performance improvement increases with the number of queries per processor. The reason for this is as follows. When the number of queries is small, the execution time is dominated by the synchronization

overhead, but when the number of queries increases, the advantage of using bulk transfers instead of automatic packet combining is accentuated.

The advantage of `bsp_hpsend` over `bsp_send` is that the message need not be buffered on the source (i.e., one `memcpy` is saved). In the left figures the curves labeled (2a) and (2b) coincide, but the right figures show that on the CC and the T3E, using `bsp_hpsend` instead of `bsp_send` indeed improves performance. On the PC cluster, however, both program version exhibit almost identical performance. This is because communication is rather slow on this architecture compared to local computations, so saving one `memcpy` does not yield a notable performance improvement.

#### 2.4. Nested parallelism and subset synchronization

In the BSP model, as implemented by e.g., the BSPlib library [30], there is no support for nested parallelism, i.e., it is not possible to split the processors into independent groups that execute asynchronously of each other. The PUB library, on the other hand, provides a partition operation that splits the processors into several subsets, each of which behaves as an autonomous BSP computer with its own synchronization points. This implies that PUB also supports subset synchronization.

Being able to create independent groups of processors has several advantages. First, as already mentioned in the introduction, in some applications completely different subalgorithms have to be performed independently in parallel. Second, many algorithms, including parallel binary search, employ a divide-and-conquer strategy in which subproblems have to be solved in parallel on smaller subsystems. Note that performing a barrier synchronization after every level of the search-butterfly prevents processors from working ahead until all processors are ready. Third, synchronization cost is typically not constant but increases with the number of processors. Subset synchronization is, therefore, faster than synchronizing all processors. Keßler [19] provides several other reasons for supporting nested parallelism.

The changes necessary to the code are shown in Program 3. The call to `bsp_partition` in line 19 partitions the processors into two groups of equal size, each of which acts as an autonomous BSP computer with its own processor numbering and message administration. In general, `bsp_partition(bsp, subbsp, n, part)` partitions the processors into  $n$  groups consisting of the processors with IDs  $[0, \text{part}[0] - 1]$ ,  $[\text{part}[0], \text{part}[1] - 1]$ , etc. It returns a pointer `subbsp` to a new BSP object of type `tbsp` which must be passed to subsequent library calls. The processors rejoin by calling `bsp_done`, after which the supergroup is reactivated.

#### Program 3. Parallel binary search with partition operations

```

1 void bin_search(pbsp bsp, int dim, int m)
2 {
3   mypid = bsp_pid(bsp); nprocs = bsp_nprocs(bsp);
4   for (i = new_m = other = 0; i < m; i++)
5     if ((query[i] <= gkey[dim] && mypid >= nprocs/2)
6         || (query[i] > gkey[dim] && mypid < nprocs/2))

```

```

7         temp[other++] = query[i];
8     else
9         query[new_m++] = query[i];
10    bsp_hpsend(bsp, (mypid+nprocs/2) % nprocs, temp,
11              other* sizeof(int));
12    bsp_sync(bsp);
13    msg = bsp_getmsg(bsp, 0);
14    memcpy(&query[new_m], bspmsg_data(msg),
15          bspmsg_size(msg));
16    new_m += bspmsg_size(msg)/sizeof(int);
17    if (dim == 0) local_search(new_m, query, n, key);
18    else {
19        part[0] = nprocs/2; part[1] = nprocs;
20        bsp_partition(bsp, subbsp, 2, part);
21        bin_search(subbsp, dim-1, new_m);
22        bsp_done(subbsp);
23    }

```

Because the processors are renumbered after a partition operation (from zero to the group size minus one), the code routing each query to the processor containing the correct subtree can also be rewritten to reflect the recursive nature of the problem. In particular, the enquiry function `bsp_pid(bsp)` returns the ID of the calling processor in the processor group associated with BSP object `bsp`, and `bsp_nprocs(bsp)` returns the number of processors in the group. Furthermore, the macro's `inLeftHalf` and `inRightHalf` have now been replaced by the tests `mypid < nprocs/2` and `mypid >= nprocs/2`, and the ID of the processor opposite to processor `mypid` in its processor group is given by `(mypid+nprocs/2) % nprocs`.

The curves labeled (3a) in Figs. 2–4 chart the running time of Program 3. On the CC and the PC cluster, Program 3 constantly outperforms Program 2, especially when the number of queries per processor is small. This can be expected since partitioning mainly reduces the synchronization time, which is most significant when the number of queries is small. On the T3E, however, Program 3 is hardly faster than either version of Program 2 when the number of queries is small (in the right panel of Fig. 4, the curves labeled (2a), (2b), and (3a) coincide and are, therefore, displayed as one curve), and it is even a little bit slower (by at most 9.7%) when the number of queries per processor is larger than 4K. On this architecture, the overhead caused by the partition operations neutralizes the reduced synchronization time.

### 2.5. Oblivious synchronization

In many problems, including parallel binary search, the programmer knows exactly how many messages each processor is due to receive. In these cases, the cost of a barrier synchronization can be avoided altogether by using *oblivious synchronization*.

For this, PUB provides the function `bsp_oblsync(bsp, n)`, which simply blocks the calling processor until `n` messages have been received. The BSP semantics are preserved by numbering the supersteps. This is explained in detail in Section 4.2.

The curves labeled (3b) in Figs. 2–4 depict the running times of Program 3 when `bsp_oblsync` is used instead of `bsp_sync`. On the CC as well as the PC cluster, this program variant is the fastest (except for one input size on the PC cluster). Especially for small instances, performance has improved significantly, since then the synchronization overhead constitutes a large fraction of the total execution time. On the T3E, the version of Program 3 that uses `bsp_oblsync` is indeed faster than the version that uses `bsp_sync`, but when there are more than 4K queries per processor, it is outperformed by the “completely synchronous” variants.

### 3. Example 2: Sorting

In this section we illustrate some of the other programming primitives provided by the PUB library using sorting as an example. The sorting algorithm employed is similar to a sorting algorithm due to Nassimi and Sahni [25]. It sorts  $\sqrt{p}$  keys by comparing each key with every other key, computing their ranks in the final order, and by routing the key with rank  $r$  to the processor with ID  $r$ . Although not the most efficient sorting algorithm in practice, it is a useful example because it concisely demonstrates some of PUB’s features. In particular, it illustrates some of PUB’s powerful collective communication routines.

In order to describe the algorithm, it is convenient to think of the processors as being organized in a  $\sqrt{p} \times \sqrt{p}$  mesh with row-major indexing. Initially, the processors in row 0 (processor 0 to  $\sqrt{p} - 1$ ) each contain one key, and finally, processor  $i$  ( $0 \leq i < \sqrt{p}$ ) holds the key with rank  $i$ . The algorithm consists of three supersteps. In the first superstep, each processor  $i$  ( $0 \leq i < \sqrt{p}$ ) broadcasts its key to all processors in row  $i$  as well as column  $i$ . In the second superstep, the processor in row  $i$  and column  $j$  compares the two keys it received in the previous superstep and sets a local variable *rank* to 1 (0) if the key it received from processor  $j$  is larger (smaller) than the key it received from processor  $i$ . After that, the *rank*-values in each column are summed and the results are stored in the processors in the last row, so that afterwards, these processors contain the final ranks of each key. In the third and final superstep, each processor in the last row send its keys to the correct output location. The algorithm will be referred to as XY-Sort.

#### 3.1. Collective communication operations

Program 4 shows how XY-Sort can be implemented using PUB. In the first superstep (lines 3–9), each processor  $i$  ( $0 \leq i < \sqrt{p}$ ) broadcasts its key to all processors in row  $i$  as well as column  $i$ . The macro’s `FirstOfRow(mypid)` and `LastOfRow(mypid)` return the IDs of the first and last processor in row `mypid`, respectively. The routine `bsp_gsend(bsp, first, last, buf, size)` broadcasts the data of length `size` pointed to by `buf` to the processors with consecutive IDs

first, first+1, ..., last. The broadcast to all processors in a column cannot be performed using `bsp_gsend`, because these processors are not numbered consecutively. Instead, the function `bsp_lsend(bsp, table, n, buf, size)` is used, which broadcasts the data pointed to by `buf` to the processors `table[0]`, `table[1]`, ..., `table[n-1]`.<sup>1</sup>

In contrast to the way global communication operations are implemented in most message-passing libraries, in PUB they are non-blocking. It is, therefore, possible to initiate several broadcast and/or parallel prefix operations in the same superstep. The results of a global communication operation are only available after the next synchronization barrier. Furthermore, as the example illustrates, PUB's collective communication operation can be used on any arbitrary subset of processors. In many other message-passing libraries, all processors have to participate.

#### Program 4. XY-Sort

```

1 void XY_Sort(void)
2 {
3   for (i = 0; i < sqrt_p; i++)
4     dst[i] = i*sqrt_p + myCol;
5   if (myRow == 0) {
6     bsp_gsend(bsp, FirstOfRow(mypid), LastOfRow(mypid),
7              &key, sizeof(int));
8     bsp_lsend(bsp, dst, sqrt_p, &key, sizeof(int));
9   }
10  bsp_oblsync(bsp, 2);
11  msg = bsp_findmsg(bsp, myCol, 0);
12  key1 = *(int *)bspmsg_data(msg);
13  msg = bsp_findmsg(bsp, myRow, 0);
14  key2 = *(int *)bspmsg_data(msg);
15
16  rank = (key1 > key2 || (key1 == key2 && myCol > myRow)) ? 1 : 0;
17
18  bsp_lreduce(bsp, dst, sqrt_p, &rank, &rank, 1,
19             bspop_addint, NULL);
20  bsp_push_reg(bsp, &key, sizeof(key));
21  bsp_oblsync(bsp, 1);
22
23  if (mypid >= p-sqrt_p)
24    bsp_put(bsp, rank, &key1, &key, 0, sizeof(key));
25  bsp_pop_reg(bsp, &key);
26  bsp_sync(bsp);
27 }
```

<sup>1</sup> As an aside, the name `gsend` stands for *group send*, and `lsend` stands for *list send*.

Because each processor receives exactly two messages in the first superstep of Program 4, we can again use `bsp_oblsync` to terminate the first superstep. After that, each processor contains two keys, one originating from the first processor in its column (processor `myCol`), the other coming from processor `myRow`. The calls to `bsp_findmsg(bsp, src, i)` in lines 11 and 13 return a message handle to the  $i$ th message in the message queue coming from the specified source. If such a message does not exist, a `NULL` pointer is returned. After that, each processor compares its two keys and sets `rank` to 1 (0) if the first key is larger (smaller) than the second key. Ties are broken by using the original position of each key as a secondary key.

At this point, the final rank of each key can be computed by summing the rank-values in every column. This is done using `bsp_lreduce(bsp, table, n, sbuf, rbuf, count, op, param)`. This function performs a reduction operation on the elements contained in `sbuf`, and stores the results in `rbuf`. The parameter `op` specifies the (associative) operation to be performed and `count` contains the size of the arrays `sbuf` and `rbuf`. Furthermore, only the processors with IDs `table[0]` to `table[n-1]` participate in the reduction, and the results are stored in processor `table[n-1]`. Note that this is a vector reduction operation, meaning that the reduction operation is performed on corresponding elements in the `sbuf` array. The void pointer `param` can be used to pass a parameter to the operation `op`.

Besides the collective operations described here, PUB also supports unbuffered high-performance variants of `bsp_gsend` and `bsp_lsend`, as well as all-to-all versions of the reduction and scan operations where all processors receive the result. For a complete list of all primitives provided by PUB, the reader is referred to [5].

After the second superstep, the processors in the last row contain the rank of each key. The final step is to route the key with rank  $r$  to the processor with ID  $r$ . This is done using DRMA.

### 3.2. Direct remote memory access

Lines 19-24 in Program 4 illustrate how DRMA operations are used in PUB. PUB's DRMA operations are identical to those provided by BSPLib [15], except for the addition of the parameter `bsp`. The call `bsp_push_reg(bsp, ident, size)` registers the local memory area starting at address `ident` for remote memory access. The `size` parameter denotes the size (in bytes) of the area being registered. This registration is required because the variable `ident` is not necessarily stored at the same address on every processor.

In line 23, the processors in the last row write the keys to their correct output locations. This is done by calling `bsp_put(bsp, pid, src, dest, offset, size)`, which copies the local memory area pointed to by `src` to the local memory block starting at address `dest+offset` of processor `pid`. The size (in bytes) of the memory area being copied is specified by the parameter `size`. Finally, the call `bsp_pop_reg(bsp, ident)` de-registers a previously registered local memory area starting at address `ident`.

Besides `bsp_put`, PUB also provides the operation `bsp_get`, which reads from the local memory of another processor, as well as unbuffered variations of these operations (`bsp_hpput` and `bsp_hpget`).

#### 4. Additional features and implementation details

In this section we describe some features of PUB that were somewhat underexposed in the previous sections and provide some implementation details.

##### 4.1. BSP Objects

As the examples have already shown, the first parameter of most functions provided by PUB is a pointer to a structure of type `tbsp`. These *BSP objects* serve three purposes. First, they are used to distinguish different groups of processors. Second, they are used for modularity and safety reasons. Third, they can be used to interleave different BSP computations running on the same processor. We now describe these objectives in detail.

###### 4.1.1. Subgroups

As shown in Section 2, one is able to create independent processor groups using the operation `bsp_partition`. This function returns a pointer to a new BSP object, which contains information like the IDs of the first and last processor in the group (to calculate the “real” ID of the destination processor), an integer identifying the object, and a pointer to the old BSP object. After a partition operation, each subgroup acts as an autonomous BSP computer with its own processor numbering, message buffer and synchronization points. Subgroups have to be created and removed in a stack-like (last-in-first-out) order. Because of this, `bsp_partition` and `bsp_done` incur no communication. A new BSP object is obtained by simply copying the old one, adjusting the values representing the IDs of the first and last processor, and by increasing the integer identifying the object by one.

###### 4.1.2. Modularity

The second use of BSP objects has to do with modularity and safety. Suppose a subroutine (e.g., `sort`) is made available in a parallel library. If no safety measures are taken, messages sent before `sort` is invoked might interfere with messages sent in `sort` or vice versa. To be able to prevent this without having to synchronize just before and just after `sort` is called, PUB provides the operation `bsp_dup`. This operation essentially duplicates the current BSP object, but with a new integer identifying the object. So, if the first (resp. last) statement of `sort` is `bsp_dup` (resp. `bsp_done`), there is no need to synchronize, because messages sent before `sort` is called cannot be taken away by `sort`, but will be available only after the first barrier synchronization after `sort`. This mechanism can be compared with MPI’s *communicators*. `bsp_dup` must also be called in a stack-like order and, therefore, incurs no communication.

#### 4.1.3. Interleaving BSP computations

To explain the third use of BSP objects, consider a parallel adaptive finite element application [7]. This application discretizes the domain into a mesh of finite elements. Each processor works on a portion of the mesh and may adaptively refine the mesh if necessary. Because it is generally impossible to predict how many new mesh points are generated and where, a load balancing algorithm is executed in a separate thread, which continuously compares the load on the processor with the load on other processors. If the load imbalance exceeds a certain threshold, the calculation is interrupted and a load balancing step is invoked.

For such adaptive applications, PUB provides the operation `bsp_thread-safe_dup`. This function returns a pointer to a new BSP object, which can be used even in different threads. This means that messages sent in one thread do not interfere with messages sent in other threads, and that barrier synchronizations executed in one thread do not suspend the processor but only the threads associated with the new BSP object. In other words, it provides a way to interleave different BSP computations running on the same processor. Because all processors have to agree on a common number identifying the new BSP object, `bsp_threadsafe_dup` incurs communication, in contrast to `bsp_dup` and `bsp_partition`. This is necessary because the number of newly created BSP objects can be different on each processor.

## 4.2. Implementation details

### 4.2.1. Overlapping communication with computation

In Section 2 we mentioned that PUB automatically combines small packets into larger ones. Packet combining is also performed in BSPlib [30] but our implementation differs in the following way. In BSPlib, *all* communication is postponed until the end of the superstep and all packets destined for the same processor are sent in a single large message (on some architectures, data communicated using `bsp_hput` or `bsp_hget` are not combined). In the implementation of PUB, each processor has  $p - 1$  output buffers, one for each destination. Small packets are not sent immediately, but instead the packet payload and some header information are written into the buffer associated with the destination processor. If the buffer is full, the data contained in the buffer is sent to its destination. Larger messages sent using one of the unbuffered point-to-point communication primitives, however, bypass the output buffer. In this way, packet combining can be performed, whilst still overlapping communication with local computations. In contrast to BSPlib implementations [30], the implementation of PUB does not rearrange the order in which messages are sent (i.e., BSPlib changes the order in which the combined messages are transmitted, not the order of the individual packets within a combined message).

### 4.2.2. Multithreading

On systems where threads are relatively cheap, the implementation of PUB uses multithreading in order to overlap communication with local computations and in order to provide non-blocking collective communication operations. On these platforms, the implementation starts a send and receive thread in addition to the main thread.

The main thread is simply the user thread. It performs computations and initiates message transmissions, synchronizations, etc. The send thread is sleeping on a semaphore until there is a message to send. As explained above, short messages are combined but larger messages sent using one of the unbuffered communication primitives bypass the output buffer. In the latter case, the main thread simply passes a pointer to the message to the send thread. On the receiving side, the receive thread consumes messages from the network and, depending on the type of the message, determines the action to be taken. For example, if a message is an ordinary message sent using `bsp_[hp]send` it is put in the message queue associated with the correct BSP object. On the other hand, if a broadcast message arrives, the receive thread also forwards it to other processors if necessary.

On platforms where no threads are available (for example, on the IBM SP2 the thread library and the communication library cannot be linked simultaneously to the main program) or where threads are expensive, PUB's communication operations are implemented on top of asynchronous send operations. This implies, however, that all receive operations will be delayed until the next barrier synchronization, since no processor is willing to receive during a superstep.

#### 4.2.3. Oblivious synchronization

When oblivious synchronization is employed, it can happen that a message arrives at its destination prior to the superstep in which it should arrive. Consider, for example, the following scenario. Processor 0 exchanges one message with processor 1 after which they synchronize by calling `bsp_oblsync(bsp, 1)` and, in the same superstep, processor 2 exchanges 100 messages with processor 3 and they synchronize by calling `bsp_oblsync(bsp, 100)`. After that, processor 0 sends one message to processor 2. Then, it can happen that this last message arrives at processor 2 before processor 2 received all messages from processor 3. To eliminate such race conditions, the supersteps are numbered. If a message with an invalid superstep number arrives, the receive thread puts it in a wait queue, which is scanned if a new superstep commences.

#### 4.3. Collective communication

The “best” way to implement a collective communication operation such as a broadcast is highly machine dependent since it depends on the size and topology of the architecture, the startup cost of a message transmission, whether the processors have several communication ports, etc. [11]. These architectural features are not visible nor available to a user of the BSP model (they are left out for the sake of portability), so algorithms developed under the BSP model for such global operations will most likely not attain the highest performance. That is why we decided to provide primitives for frequently occurring collective communication operations.

PUB's global operations are implemented by dynamically selecting an algorithm that maximizes performance. At configuration time, the system measures some basic machine parameters like the link bandwidth and the startup cost of a message transmission, and this information is used at run time to select the algorithm that

minimizes the time required to perform the operation. This decision is also based on the number of processors participating. The Oxford BSP toolset implementation of BSPlib [15] also dynamically chooses between different implementations of its collective communication routines, but bases its decision on the BSP parameters  $g$ ,  $L$ , and  $p$ . PUB, on the other hand, also exploits properties not visible to a user of the BSP model, such as the startup cost of a message transmission.

To illustrate the importance of providing primitives for frequently occurring collective communication operations, Fig. 5 compares the performance of PUB's broadcast routine `bsp_gsend` with three broadcast algorithms the user might implement. The user implementations are realized using PUB's basic BSP primitives `bsp_hpsend` and `bsp_sync`. The  $x$ -axis shows the size of the message to be broadcast (in bytes); the  $y$ -axis shows the time divided by the message length. The algorithm implemented on top of the core operations are *Tree*, *PTree*, and *Sca-Ga*. Algorithm *Tree* broadcasts the message in a tree-like fashion but with a  $d$ -ary instead of a binary tree (a BSP description of this algorithm is given in [10]). The optimal value of  $d$  was determined experimentally. *PTree* is a pipelined version of *Tree*. It divides the broadcast message into  $k$  messages of equal size and sends these messages along the branches of the tree in a pipelined fashion [4]. In the experiments a binary tree was used, and the optimal value of  $k$  was determined empirically. Finally, *Sca-Ga* is the scatter-gather approach described in [18]. It divides the broadcast message into  $p$  submessages of equal size which are distributed among the processors. After that, each processor sends the submessage it contains to every other processor. A more detailed description of the algorithms as well as their analysis and additional experiments are given in [26].

According to the BSP cost model, the *Sca-Ga* algorithm should be the fastest algorithm (it is optimal to within a constant factor of two). However, looking at Fig. 5 we observe that *Sca-Ga* is the fastest algorithm only when the broadcast message is larger than or equal to 16K. For messages smaller than 16K, *Tree* is the fastest algorithm, which according to the BSP model, should always be less efficient than the other two algorithms. The main reason is that the BSP model ignores the startup

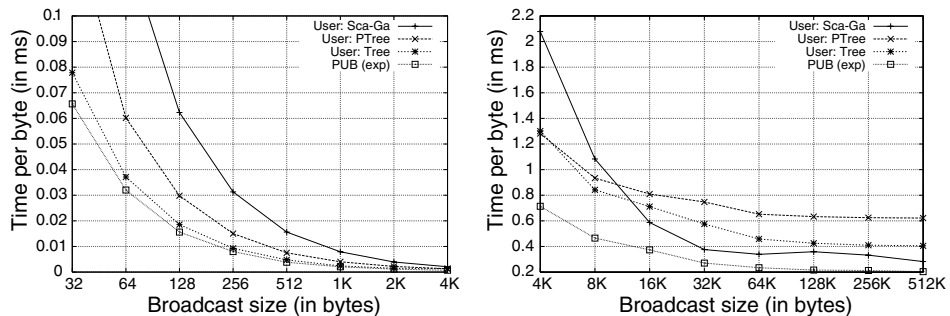


Fig. 5. Performance comparison of various broadcast algorithms on the PC cluster. The curve labeled “PUB (exp.)” depicts the performance of PUB's `bsp_gsend` routine. The other three algorithms are implemented on top of PUB's basic BSP operations.

cost of a message transmission. Because *Sca-Ga* incurs many startups, it is actually less efficient than *Tree* for small message sizes.

The algorithm selected by PUB always outperforms the three algorithms implemented on top of the core operations. In fact, PUB selects the *Tree* algorithm for small messages (up to 8K), and *Sca-Ga* for large messages. There are two reasons why PUB's broadcast is actually more efficient than implementations of these algorithms on top of the core operations. First, the user implementations do not use oblivious synchronization, since it is not available in the original BSP model. Second, they also contain overhead for retrieving the message from the message buffer and forwarding them to other processors.

Summarizing, we find that the BSP model is not detailed enough to accurately predict the running times of low-level communication primitives. Furthermore, even if the programmer is aware of this, he or she has to put a lot of effort in implementing these algorithms and parameter tuning in order to determine the fastest algorithm in practice. We, therefore, conclude that it is beneficial to provide these collective communication operations as primitives.

#### 4.4. Simulator

Developing parallel programs on the actual target architecture is often a laborious task. The programming environment may be uncomfortable because there is no adequate debugger, the time to access the parallel machine may be limited, and starting parallel programs often takes some time. For these reasons, we developed a simulator for the PUB library, which can be used to develop PUB programs on a PC or workstation. The simulator also produces performance estimates based on the BSP\* cost model [4], which is an extension of the BSP model that incorporates the startup cost of a message transmission.

### 5. Conclusions and future work

We have presented the PUB library. It is based on the BSP model but has a number of additional features, such as support for nested parallelism, oblivious synchronization, and non-blocking collective communication operations. Support for nested parallelism was included primarily because it is inconvenient to implement algorithms following a parallel divide-and-conquer paradigm in a “flat” BSP library. Moreover, as was shown in this paper, on some architectures employing nested parallelism leads to better performance, mainly because synchronizing a subset of processors is faster than synchronizing all processors. In cases where it is known how many messages each processor is due to receive, the cost of a barrier synchronization can be avoided altogether by using oblivious synchronization. We have also shown that it is useful to provide collective communication primitives, even though they can be implemented on top of the basic operations, because the library implementation can exploit architectural features not captured by the BSP model.

In our experience, frequent message buffering contributes significantly to the communication overhead. That is why PUB supports buffered as well as unbuffered variants of all its communication primitives. The buffered variants provide maximum safety, so it is advisable to use these during development. They can be replaced by unbuffered variants after making sure that the data is not altered during the current superstep.

Future work will include porting PUB to other platforms. Although PUB can be directly ported to any platform on which MPI is available, the performance will be less than when it is implemented on top of the native message-passing library. We are also developing a version of PUB that can be used on heterogeneous systems in which the processors run at different speeds. The idea is to let the user create more processes than processors and the system to take care of load balancing, process migration, etc. Another topic for future research is the integration of parallel algorithms and data structures in the PUB library. In sequential computing there are several libraries that provide the user with basic data structures like lists, trees and queues, but in parallel computing they are lacking. The final goal is to provide a library of efficient parallel algorithms on top of the PUB library.

The PUB library can be freely downloaded from [www.uni-paderborn.de/~pub](http://www.uni-paderborn.de/~pub).

## References

- [1] M. Adler, Asynchronous shared memory search structures, in: Proc. Symp. on Parallel Algorithms and Architectures 1996.
- [2] R.D. Alpert, J.F. Philbin, cBSP: Zero-Cost Synchronization in a Modified BSP Model. Technical Report 97-054, NEC Research Institute, 1997.
- [3] R. Barriuso, A. Knies, SHMEM User's Guide, Revision 2.0, 1994.
- [4] A. Bäumker, W. Dittrich, F. Meyer auf der Heide, Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model, *Theoretical Computer Science* 203 (1998).
- [5] O. Bonorden, J. Gehweiler, P. Olszta, R. Wanka, PUB-Library Version 8.0, 2002. Available via [www.uni-paderborn.de/~pub](http://www.uni-paderborn.de/~pub).
- [6] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, Parallel programming in Split-C, in: Proc. Supercomputing, 1993.
- [7] R. Diekmann, D. Meyer, B. Monien, Parallel decomposition of unstructured FEM-Meshes, in: Proc. IRREGULAR. Springer LNCS 980, 1995.
- [8] A. Fahmy, A. Heddaya, Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk. Technical Report BU-CS-96-012, Boston Univ., 1996.
- [9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM 3 Users Guide and Reference Manual. Oak Ridge National Laboratory, 1994.
- [10] A.V. Gerbessiotis, L.G. Valiant, Direct bulk-synchronous parallel algorithms, *Journal of Parallel and Distributed Computing* 22 (2) (1994).
- [11] A. Goldman, D. Trystram, J. Peters, Exchange of messages of different sizes, in: Proc. IRREGULAR. Springer LNCS 1457, 1998.
- [12] J.A. Gonzalez, C. Leon, F. Piccoli, M. Printista, J.L. Roda, C. Rodriguez, F. de Sande, Groups in bulk synchronous parallel computing, in: Proc. EuroMicro Workshop on Parallel and Distributed Processing, 2000.
- [13] J.A. Gonzalez, C. Leon, F. Piccoli, M. Printista, J.L. Roda, C. Rodriguez, F. de Sande, Oblivious BSP, in: Proc. Euro-Par. Springer LNCS 1900, 2000.

- [14] M.W. Goudreau, K. Lang, S.B. Rao, T. Tsantilas, The Green BSP Library. Technical Report TR-95-11, University of Central Florida, Orlando, 1995.
- [15] J.M.D. Hill, W.F. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, R. Bisseling, BSPlib: the BSP programming library, *Parallel Computing* 24 (14) (1998).
- [16] B.H.H. Juurlink, Experimental validation of parallel computation models on the Intel Paragon, in: *Proc. IPPS/SPDP, 1998*. Full version TR-RSFB-98-055, Paderborn University.
- [17] B.H.H. Juurlink, I. Rieping. Performance relevant issues for parallel computation models, in: *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications, 2001*.
- [18] B.H.H. Juurlink, H.A.G. Wijshoff, Communication primitives for BSP computers, *Information Processing Letters* 58 (6) (1996).
- [19] C.W. Keßler, NestStep: nested parallelism and virtual shared memory for the BSP model, in: *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications, 1999*.
- [20] J.-S. Kim, S. Ha, C.S. Jhon, Relaxed barrier synchronization for the BSP model of computation on message-passing architectures, *Information Processing Letters* 66 (1998).
- [21] W.F. McColl, Universal computing, in: *Proc. Euro-Par. Springer LNCS 1123, 1996*.
- [22] Message Passing Interface Forum. MPI: a Message passing interface, in: *Proc. Supercomputing, 1993*.
- [23] Message Passing Interface Forum. MPI-2: Extension to the Message Passing Interface. Technical report, Univ. of Tennessee, 1997.
- [24] R. Miller, A library for bulk synchronous parallel programming, in: *Proc. BCS Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing, 1993*.
- [25] D. Nassimi, S. Sahni, Parallel permutation and sorting algorithms and a new generalized connection network, *Journal of the ACM* 29 (3) (1982).
- [26] I. Rieping, *Communication in Parallel Systems—Models, Algorithms and Implementations*. PhD thesis, Paderborn University, 2000.
- [27] F. de Sande, *El Modelo de Computación Colectiva: Una Metodología Eficiente para la Ampliación del Modelo de Librería de Paso de Mensajes con Paralelismo de Datos Anidado*. PhD thesis, University of La Laguna, Spain, 1998. (In Spanish).
- [28] J.P. Singh, J.L. Hennessy, Finding and exploiting parallelism in an ocean simulation program: experience, results and implications, *Journal of Parallel and Distributed Computing* 15 (1) (1992).
- [29] D.B. Skillicorn, miniBSP: a BSP Language and Transformation System. Technical report, Dept. of Computing and Information Sciences, Queens' University, Kingston, Canada, 1996.
- [30] D.B. Skillicorn, J.M.D. Hill, W.F. McColl, Questions and answers about BSP, *Journal of Scientific Programming* 6 (3) (1997).
- [31] L.G. Valiant, A bridging model for parallel computation, *Communications of the ACM* 33 (8) (1990).