In: 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP), San Juan, Puerto Rico, April 12 - April 16, 1999

The Paderborn University BSP (PUB) Library – Design, Implementation and Performance

Olaf Bonorden[1] Ben Juurlink[2] Ingo von Otte[1] Ingo Rieping[1]

Abstract

The Paderborn University BSP (PUB) library is a parallel C library based on the BSP model. The basic library supports buffered and unbuffered asynchronous communication between any pair of processors, and a mechanism for synchronizing the processors in a barrier style. In addition, it provides routines for collective communication on arbitrary subsets of processors, partition operations, and a zero-cost synchronization mechanism. Furthermore, some techniques used in its implementation deviate significantly from the techniques used in other BSP libraries.

1. Introduction

Most message-passing libraries, like PVM[6] and MPI[12], 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. 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 complexity of programs implemented using pairwise sends and receives.

An alternative parallel programming model is provided by the BSP (Bulk Synchronous Parallel) model[15]. BSP computations consists of a sequence of *supersteps*, separated by barrier synchronizations. In every superstep, each processor can perform local operations and send some messages, which are assured to have reached their destinations at the beginning of the next superstep. This approach is less prone to deadlocks since there are no explicit receives. Instead, a barrier synchronization signifies the end of all communication operations. Moreover, BSP 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, see [15, 11].

In this paper we present the Paderborn University BSP (PUB) library. It has three features that distinguishes it from other BSP libraries. First, it provides a rich set of collective communication operations like broadcast, reduce, and scan, in addition to basic BSP functionality. Second, it provides the possibility to dynamically partition the processors into several subsets, each of which acts as an autonomous BSP computer with its own processor numbering and synchronization points. Third, it introduces the concept of *BSP objects*, which are used for (1) distinguishing different processor groups, (2) modularity and safety purposes, and (3) assuring that messages sent in different threads do not interfere with each other.

Currently, the PUB library is available on several Parsytec systems, the Intel Paragon, the IBM SP2, and clusters of workstations. An implementation on top of MPI is available too. There is also a simulator so that PUB programs can be developed and tested on a PC or workstation.

Related Work. The Oxford BSP library[13] was the first BSP library. Basically, it contains functions for delimiting supersteps and Direct Remote Memory Access (DRMA). Only static variables can be accessed through DRMA.

The Green BSP library[7] does not provide DRMA but only one Bulk Synchronous Message Passing (BSMP) operation, that sends a fixed-size packet to the specified destination processor. There is no receive operation, but instead the message is stored in a queue, and is available only after the next barrier synchronization. In addition, it offers several enquiry functions that are used to determine the number of processors, the number of messages in the queue, etc.

BSPlib[9] provides DRMA as well as BSMP operations. Moreover, the restriction that only static variables could be accessed through DRMA operations is eliminated by requiring that variables are first registered. PUB offers the same functionality as BSPlib, but in addition provides several other features. In fact, the PUB library can be configured so that it accepts programs written using BSPlib.

Several authors (e.g., [2, 5]) recognized that a barrier synchronization is expensive on many parallel architectures

[1]Heinz Nixdorf Institute and Dept. of Comp. Sci., Paderborn University. Fürstenallee 11, 33102 Paderborn, Germany.
E-mail {bono, ivo, inri}@uni-paderborn.de
Supported in part by DFG-SFB 376 "Massive Parallelität" and EU ESPRIT Long Term Research Project 20244 (ALCOM-IT).
[2]Laboratory of Computer Architecture and Digital Techniques, Dept. of Electrical Engineering, Delft Technical University. Mekelweg 4, 2628 CD Delft, The Netherlands.
E-mail B.H.H.Juurlink@its.tudelft.nl

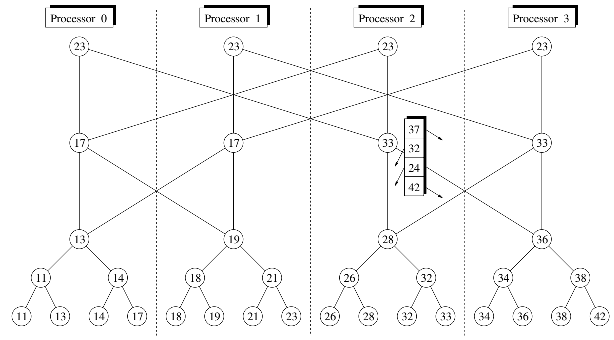

Figure 1. Search-butterfly with $\mathbf{n = m = p = 4}$.

and modified the BSP model to include a zero-cost synchronization mechanism, which can be used if the number of messages each processor is due to receive is known. It is also available in PUB, and we call it *oblivious synchronization*.

The full version of this paper is available as a technical report[3].

2. An Example: Binary Search

In order to describe the PUB library, we interweave examples with library functions descriptions and performance results. This section describes most of PUB's message-passing routines using parallel binary search as an example.

Parallel Binary Search. In parallel binary search one has to locate $M = m{\cdot}p$ *queries* in a search structure of $N = n{\cdot}p$ keys. We use a search-butterfly[1], in which the root of the search tree is replicated p times and every other level 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 and illustrates how it is distributed.

To locate the queries, they first have to be routed to the correct subtree. This is done by sending them to the left or right half at every level of the butterfly, depending on whether the query is smaller than the current node's key or not, as illustrated in Fig. 1. We say that processor P belongs to the left (right) half of the butterfly in dimension d, if the d-th least significant bit in its binary representation is 0 (1).

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 code in lines 3-9 routes each query to its correct subtree. Here, `inLeft(d,me)` and `inRight(d,me)` are macro's that return 1 if processor `me` belongs to the left or right half of the butterfly in dimension `d`, respectively, and `Opposite(d,me)` is a macro that returns the ID of the processor opposite to processor `me` in dimension `d`. The call `bsp_send(bsp,dest,buf,size)` in line 6 sends a message of `size` bytes to processor `dest`. The first argument `bsp` is a pointer to an object of type `tbsp`, which must be passed to most PUB functions.

```
 1 void bin_search(int d, int m)
 2 {
 3   for (i = new_m = 0; i < m; i++)
 4     if (query[i]<=gkey[d] && inRight(d,me) ||
 5         query[i]>gkey[d] && inLeft(d,me))
 6         bsp_send(bsp, Opposite(d,me),
 7                  &query[i],sizeof(int));
 8     else
 9         query[new_m++] = query[i];
10   bsp_sync(bsp);
11
12   for (i = 0; i < bsp_nmsgs(bsp); i++)
13   {
14     msg = bsp_getmsg(bsp,i);
15     query[new_m++]= (int)(*bspmsg_data(msg));
16   }
17   if (!d) local_search(new_m, query, n, key);
18   else bin_search(d-1, new_m);
19 }
```

Program 1: Initial parallel binary search program.

The call `bsp_sync(bsp)` performs a barrier synchronization. When it returns, all messages are available in a message queue at their destinations. The size of this queue can be determined by calling `bsp_nmsgs`, and messages can be fetched by calling `bsp_getmsg(bsp,i)`. This function returns a message handle `msg` to the `i`-th message in the queue, but does not remove it from the queue (it exist until the next synchronization point). In contrast, in BSPlib messages can only be removed from the front of the queue. We decided to provide random access into the message queue, because first, this is more flexible, and second, messages do not always need to be copied.

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 its size (`bspmsg_size(msg)`) and the ID of the source (`bspmsg_src(msg)`). As these functions operate on messages, no BSP object has to be supplied. Finally, in lines 17-18, 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 combine small packets into larger ones, similar to the way it is done in many implementations of BSPlib[14]. In these BSPlib implementations, all communication is postponed until the end of the superstep and all packets destined for the same processor are sent as one. This means, however, that there can be no overlap of communication and computation. In our implementation, each processor has p output buffers. Small packets are not sent immediately, but instead the packet payload and some header information is written into the correct

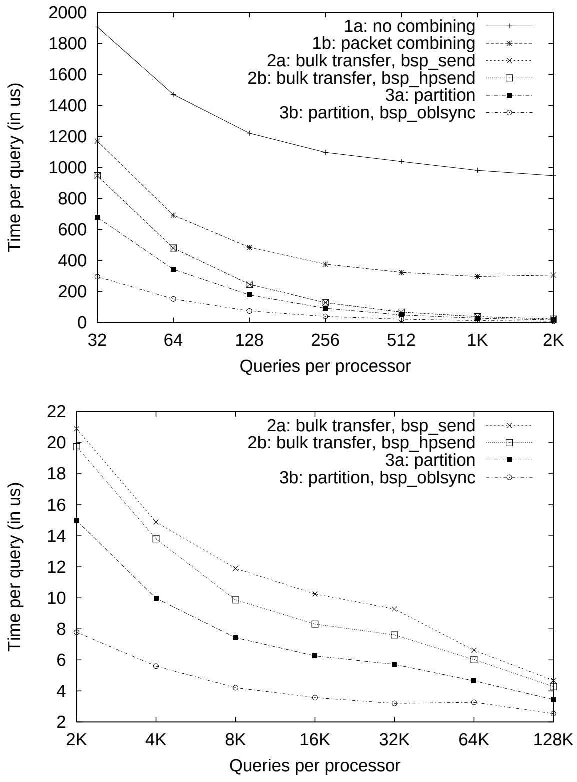

Figure 2. Time per query for several versions of parallel binary search on a Parsytec CC.

```
void bin_search(int d, int m)
{
  for (i=new_m=other=0; i<m; i++)
    if (query[i]<=gkey[d] && inRight(d,me) ||
        query[i]>gkey[d] && inLeft(d,me))
        temp[other++] = query[i];
     else
        query[new_m++] = query[i];

  bsp_hpsend(bsp, Opposite(d,me), temp,
             other*sizeof(int));
  bsp_sync(bsp);

  msg = bsp_getmsg(bsp,0);
  memcpy(&query[new_m], bspmsg_data(msg),
         bspmsg_size(msg));
  new_m += bspmsg_size(msg)/sizeof(int);
  if (!d) local_search(new_m, query, n, key);
  else bin_search(d-1, new_m);
}
```

Program 2: Parallel binary search with bulk transfers.

buffer. Messages larger than the buffer are sent instantaneously. If the buffer is full, the data contained in the buffer is sent to its destination. In this way, packet combining can be performed, whilst still overlapping communication with local computations.

Fig. 2 compares the performance achieved by Program 1 without (curve (1a)) and with (1b) packet combining on a 32-processor Parsytec CC. The x-axis shows the number of queries per processor m; the y-axis shows the time per query per processor needed for routing the queries to their subtrees. The time taken by the local search is not included. It can be seen that packet combining improves performance significantly.

Bulk Transfers. Sending many small packets still has a negative impact on performance, however, because several bytes of header information have to be sent with every packet and because packing/unpacking is not for free[10]. To be able to use bulk transfer in our example, the programmer must pack the data, as shown in Program 2. We also replaced `bsp_send` by `bsp_hpsend`. This function is unbuffered, meaning that the data should not be changed until the next synchronization. This reduces data copying.

Curve (2a) in Fig. 2 depicts the performance achieved by Program 2 using `bsp_send` and curve (2b) shows the performance attained using `bsp_hpsend`. The performance has improved significantly compared to Program 1. When the number of queries is small, the gain is moderate, because then the execution time is dominated by the synchronization overhead. But as the number of queries increases, the advantage of using bulk transfers becomes more significant. The bottom figure shows that using `bsp_hpsend` also improves performance. However, because the data still has to be bundled, it improves performance only slightly. The relative benefits are larger when the message payload is already stored contiguously.

Subgroups. PUB offers an operation that partitions the processors into several independent subsets. Consequently, PUB supports subset synchronization. Being able to do this has several advantages. First, sometimes different subalgorithms have to be executed in parallel. Second, many algorithms, including binary search, employ a divide-and-conquer strategy. Performing a barrier synchronization after every level of the search-butterfly prevents the 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.

Program 3 shows the changes necessary to the code. The call to `bsp_partition` in line 22 partitions the processors into two groups of equal size. In general, `bsp_partition(bsp,subbsp,n,part)` partitions the processors into n groups consisting of the processors $[0, \texttt{part}[0]-1]$, $[\texttt{part}[0], \texttt{part}[1]-1]$, etc., and returns a pointer `subbsp` to a new BSP object of type `tbsp`. The processors rejoin by calling `bsp_done`.

Because the processors are renumbered after a partition operation (from 0 to the group size minus one), the code routing each query to its subtree is also simplified. In particular, the enquiry function bsp_pid(bsp) returns the ID of the calling processor in the group associated with BSP object bsp, and bsp_nprocs(bsp) returns the number of processors in the group. Furthermore, the macro's inLeft and inRight have been replaced by the tests me<nprocs/2 and me>=nprocs/2, and the ID of the processor opposite to processor me in its group is given by (me+nprocs/2) % nprocs.

```
void bin_search(pbsp bsp, int d, int m)
{
  me=bsp_pid(bsp); nprocs=bsp_nprocs(bsp);
  for (i=new_m=other=0; i < m; i++)
    if (query[i]<=gkey[d] && me>=nprocs/2 ||
        query[i]>gkey[d] && me<nprocs/2)
        temp[other++] = query[i];
    else
        query[new_m++] = query[i];
  bsp_hpsend(bsp, (me+nprocs/2) % nprocs,
             temp, other*sizeof(int));
  bsp_sync(bsp);

  msg = bsp_getmsg(bsp, 0);
  memcpy(&query[new_m], bspmsg_data(msg),
         bspmsg_size(msg));
  new_m += bspmsg_size(msg)/sizeof(int);
  if (!d) local_search(new_m,query,n,key);
  else
  {
    part[0]= nprocs/2; part[1]= nprocs;
    bsp_partition(bsp, subbsp, 2, part);
    bin_search(subbsp, d-1, new_m);
    bsp_done(subbsp);
  }
}
```

Program 3: Binary search with partition operations.

Curve (3a) in Fig. 2 shows the performance achieved by Program 3. When the number of queries is small ($m \leq 64$), it has improved by about 28% compared to Program 2. When there are many queries, however, the improvement is small. This is because partitioning reduces the synchronization time.

Synchronization. The term "barrier synchronization" is traditionally used to denote a logical point in a program that all processors must reach before the computation may proceed. In the BSP model, however, it means something stronger. It also requires that all communication operations have completed. Because of this, a simple binary tree algorithm cannot be used.

In the PUB library, a barrier synchronization is implemented by a global reduction on the number of messages each processor has outstanding. Although this method works very efficiently, it causes communication overhead. In many problems, including binary search, the programmer knows how many messages each processor is due to receive. In that case, communication can be avoided altogether by using *oblivious synchronization*. For this, PUB offers the function bsp_oblsync(bsp,n), which does not return until n messages have been received.

Curve (3b) in Fig. 2 shows the performance improvement resulting from using bsp_oblsync instead of bsp_sync. Overall, we have improved the performance of parallel binary search by a factor of 4-40 compared to the initial Program 1 with packet combining.

3. Additional Features

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

3.1 Collective Communication. PUB provides several routines for collective communication on arbitrary subsets of processors. For example, the function bsp_gsend(bsp,first,last,buf,size) broadcasts the data of length size starting at buf to the processors first,...,last. If the processors are not consecutively numbered, we have to use bsp_lsend(bsp,table,n,buf,size), which broadcasts the data pointed to by buf to the processors table[0],...,table[n−1].

In contrast to the way global communications are implemented in most message-passing libraries, in PUB they are non-synchronizing. This means that the results are not available until the next synchronization barrier. It is therefore possible to initiate several broadcast or parallel prefix operations in the same superstep. Furthermore, PUB's collective communication operations can be used on any arbitrary subset of processors. In many other message-passing libraries, all processors must participate.

3.2 BSP Objects. As the example has already shown, the first parameter of most PUB functions is a pointer to a structure of type tbsp. These *BSP objects* serve three purposes, which are described below.

Subgroups. The function bsp_partition 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 "actual" IDs and to check whether an ID is legal), an integer identifying the object, and a pointer to the old BSP object. After a partition step, each subgroup acts as an autonomous BSP computer with its own processor numbering, message queue and synchronization points. Subgroups have to be created and removed in a stack-like order. Because of this, bsp_partition and bsp_done incur no communication. Instead, 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.

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. Then, for safety reasons, a barrier synchronization should be performed just before and just after `sort` is invoked, or otherwise messages sent before `sort` is called might interfere with messages sent in `sort`. The operation `bsp_dup` prevents this from happening. This operation essentially duplicates the current BSP object, but with a new integer identifying the object. So, if the first (last) statement of `sort` is `bsp_dup` (`bsp_done`), there is no need to synchronize, because messages sent before `sort` is called cannot be taken away by `sort`. This mechanism can be compared with *communicators* in MPI. `bsp_dup` must also be called in a stack-like order and, therefore, incurs no communication.

Interleaving BSP computations. To explain the third use of BSP objects, consider a parallel adaptive finite element application[4]. This application discretizes the domain into a mesh of finite elements. Each processor works on a portion of the mesh and may refine the mesh if necessary. Because it is impossible to predict how many new points are generated and where, a load balancing algorithm is executed in a parallel 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 applications, PUB provides the operation `bsp_threadsafe_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 different threads do not interfere with each other, and that barrier synchronizations executed in one thread do not suspend the processor but only the threads associated with the object. In other words, it provides a way to interleave different BSP computations running on the same processor. However, because all processors have to agree on a common number identifying the new BSP object, `bsp_threadsafe_dup` incurs communication.

3.3 Implementation Details. On systems where threads are relatively cheap, the implementation of PUB uses multithreading in order to overlap communication with computations and to provide non-blocking collective communication operations. On these platforms, PUB starts a `send` and a `receive` thread. The `send` thread is sleeping until there is a message in one of the output buffers. If so, it sends it to its destination. Recall that short messages are not sent immediately, but only once the buffer is full. Furthermore, messages sent using `bsp_hpsend` bypass the output buffer. In that case, the main program simply passes a pointer 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_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.

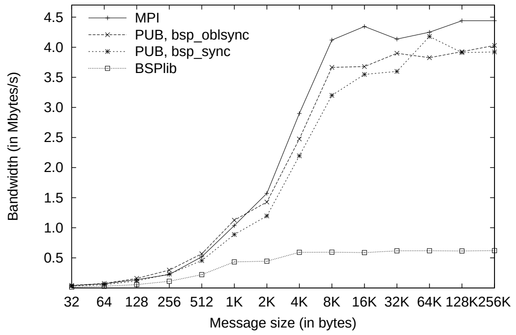

Figure 3. Communication performance of PUB, MPI, and BSPlib on an ATM cluster.

Using PUB, it can happen that a message arrives at its destination prior to the superstep in which it should arrive. Consider the following scenario. Processor P_0 exchanges one message with P_1 after which they synchronize using `bsp_oblsync(bsp,1)` and, in parallel, P_2 exchanges 100 messages with P_3 and they synchronize by calling `bsp_oblsync(bsp,100)`. After that, P_0 sends one message to P_2. Then, it can happen that this last message arrives at P_2 before this processor received all messages from P_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 begins.

On platforms where no threads are available 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 postponed until the next barrier synchronization, since no processor is willing to receive during a superstep.

4. Performance Measurements

In this section, we compare the communication performance of PUB with that of MPI and the Oxford BSP Toolset implementation of BSPlib[8]. This is done by showing the per-processor bandwidth achieved in a total exchange communication pattern as a function of the message length.

Fig. 3 shows the experimental results gathered on a cluster of 16 workstations connected by an ATM network. Two PUB programs are included; one that uses `bsp_oblsync`

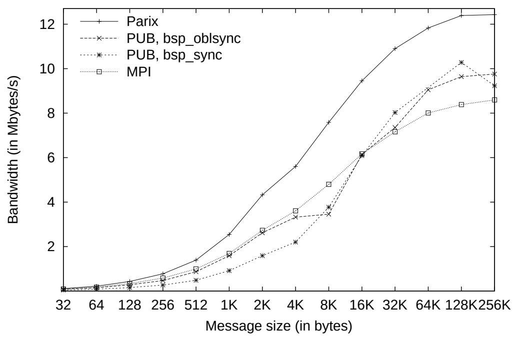

Figure 4. Communication performance of PUB, MPI, and Parix on a Parsytec CC.

and one that uses `bsp_sync`. For small messages (up to 1KB), the PUB program that uses oblivious synchronization achieves the highest performance. Even the PUB program that uses `bsp_sync` performs almost as good as MPI. It is important to note that the MPI program communicates by send-receive pairs, whereas PUB has to send additional synchronization messages. For messages larger than 1KB, MPI yields the best results, but its performance is at most 10% higher than the PUB program that uses `bsp_oblsync`. The BSPlib program performs worst. It appears that it splits messages into 2KB or 4KB packets, since the bandwidth does not increase anymore if the messages are made larger. The user guide of the Oxford BSP Toolset implementation of BSPlib[8] states that the TCP/IP version is optimized for a 10Mb/s shared Ethernet.

Fig. 4 shows the results collected on a 44-processor Parsytec CC. Because BSPlib is not implemented on this platform, we used the native communication library (Parix) instead. From Fig. 4 it can be seen that for messages up to 16KB, MPI performs slightly better than the PUB program with oblivious synchronization. For larger messages, both PUB versions attain a higher bandwidth than MPI. Whereas MPI peaks at about 8.3MB/s, PUB achieves approximately 10MB/s. Obviously, Parix achieves the highest performance. For example, for messages larger than 64KB, Parix is about 20% faster than PUB. This is because on the CC, it is beneficial to post the receive before the message arrives, in order to avoid that it needs to be buffered. In PUB, however, this mechanism cannot be used, because the source and the length of the message are unknown.

5. Concluding Remarks

We presented the PUB library; a parallel C library based on the BSP model. It is more flexible than other BSP libraries and also includes a zero-cost synchronization mechanism. This was done because PUB was developed in the project SFB 376 “Massive Parallelism”, and feedback from application developers revealed that large part of their applications operate in a bulk-synchronous fashion, but other time-critical parts do not. We, therefore, decided to develop a BSP library that adheres to the BSP model as much as possible, but add additional constructs when necessary.

References

[1] M. Adler. Asynchronous Shared Memory Search Structures. In *ACM Symp. on Parallel Algorithms and Architecture*, pages 42–51, 1996.

[2] R. Alpert and J. Philbin. cBSP: Zero-Cost Synchronization in a Modified BSP Model. Technical Report 97-054, NEC Research Institute, 1997.

[3] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library – Design, Implementation and Performance. Technical Report TR-RSFB-98-063, Heinz Nixdorf Institute, Paderborn University, 1998.

[4] R. Diekmann, D. Meyer, and B. Monien. Parallel Decomposition of Unstructured FEM-Meshes. In *IRREGULAR ’95*, 1995. LNCS 980.

[5] A. Fahmy and A. Heddaya. Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk. Technical Report BU-CS-96-012, Boston Univ., 1996.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 Users Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tenessee 37831, May 1994.

[7] M. W. Goudreau, K. Lang, S. B. Rao, and T. Tsantilas. The Green BSP Library. Technical Report TR-95-11, University of Central Florida, Orlando, 1995.

[8] J. Hill, S. Donaldson, and A. McEwan. *Installation and User Guide for the Oxford BSP Toolset (v1.3) Implementation of BSPlib*. Oxford University Computing Laboratory, November 1997.

[9] J. Hill, W. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, 1998.

[10] B. H. Juurlink. Experimental Validation of Parallel Computation Models on the Intel Paragon. In *IPPS/SPDP*. IEEE, 1998. Full version TR-RSFB-98-055, Paderborn University.

[11] W. McColl. Scalable Computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 46–61. Springer LNCS 1000, 1995.

[12] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. Supercomputing ’93*, pages 878–883. IEEE Computer Society Press, 1993.

[13] R. Miller. A Library for Bulk Synchronous Parallel Programming. In *Proceedings of the BCS Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, pages 100–108, 1993.

[14] D. Skillicorn, J. Hill, and W. McColl. Questions and Answers About BSP. *Journal of Scientific Programming*, 6(3):249–274, 1997.

[15] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), Aug. 1990.