ELSEVIER Parallel Computing 24 (1998) 1947–1980

PARALLEL COMPUTING

Practical aspects

BSPlib: The BSP programming library

Jonathan M.D. Hill [a,*], Bill McColl [a], Dan C. Stefanescu [b,c], Mark W. Goudreau [d], Kevin Lang [e], Satish B. Rao [e], Torsten Suel [f], Thanasis Tsantilas [g], Rob H. Bisseling [h]

[a] *Oxford University Computing Laboratory, Oxford OX1 3QD, UK*
[b] *Harvard University, Cambridge, USA*
[c] *Suffolk University, Boston, USA*
[d] *University of Central Florida, Orlando, USA*
[e] *NEC Research Institute, Princeton, USA*
[f] *Bell Laboratories, Lucent Technologies, N.J., USA*
[g] *Columbia University, New York, USA*
[h] *Utrecht University, Utrecht, The Netherlands*

Received 10 October 1997; received in revised form 15 April 1998

Abstract

BSPlib is a small communications library for bulk synchronous parallel (BSP) programming which consists of only 20 basic operations. This paper presents the full definition of *BSPlib* in C, motivates the design of its basic operations, and gives examples of their use. The library enables programming in two distinct styles: direct remote memory access (DRMA) using put or get operations, and bulk synchronous message passing (BSMP). Currently, implementations of *BSPlib* exist for a variety of modern architectures, including massively parallel computers with distributed memory, shared memory multiprocessors, and networks of workstations. *BSPlib* has been used in several scientific and industrial applications; this paper briefly describes applications in benchmarking, Fast Fourier Transforms (FFTs), sorting, and molecular dynamics. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Bulk synchronous parallel; Parallel communications library; One-sided communication

* Corresponding author. E-mail: jonathan.hill@comlab.ox.ac.uk

0167-8191/98/$ – see front matter © 1998 Elsevier Science B.V. All rights reserved.
PII: S 0 1 6 7 - 8 1 9 1 (9 8) 0 0 0 9 3 - 3

1. Introduction

Since the earliest days of computing it has been clear that, sooner or later, sequential computing would be superseded by parallel computing. This has not yet happened, despite the availability of numerous parallel machines and the insatiable demand for increased computing power. For parallel computing to become the normal form of computing we require a model which can play a similar role to the one that the von Neumann model has played in sequential computing. The emergence of such a model would stimulate the development of a new parallel software industry, and provide a clear focus for future hardware developments. For a model to succeed in this role it must offer three fundamental properties.

Scalability – the performance of software and hardware must be scalable from a single processor to several hundreds of processors.

Portability – software must be able to run unchanged, with high performance, on any general purpose parallel architecture.

Predictability – the performance of software on different architectures must be predictable in a straightforward way.

It should also, ideally, permit the correctness of parallel programs to be determined in a way which is not much more difficult than for sequential programs.

Recent research on Bulk Synchronous Parallel (BSP) algorithms, architectures and languages has shown that the BSP model can achieve all of these requirements [39,30,38,16,9,32].

The BSP model decouples the two fundamental aspects of parallel computation: communication and synchronisation. This decoupling is the key to achieving universal applicability across the whole range of parallel architectures. A BSP computation consists of a sequence of parallel *supersteps*. Each superstep is subdivided into three ordered phases consisting of: (1) simultaneous local computation in each process, using only values stored in the memory of its processor; (2) communication actions amongst the processes, causing transfers of data between processors; and (3) a barrier synchronisation, which waits for all of the communication actions to complete, and which then makes any data transferred visible in the local memories of the destination processes.

This approach to parallel programming is applicable to all kinds of parallel architecture: distributed memory architectures, shared memory multiprocessors, and networks of workstations. It provides a consistent, and very general, framework within which to develop portable parallel software for scalable parallel architectures.

In this work, we describe *BSPlib*, a small communications library for BSP programming in a Single Program Multiple Data (SPMD) manner. The main features of *BSPlib* are two modes of communication, one capturing a one-sided direct remote memory access (DRMA) paradigm and the other reflecting a bulk synchronous message passing (BSMP) approach.

BSPlib is not the only communication library for parallel computing. One prominent alternative is the Message Passing Interface (MPI) [34,19].

MPI and *BSPlib* are similar in that they are both designed for the development of scalable and portable parallel code. The fundamental difference between the two is that *BSPlib* is based on the superstep programming discipline, while MPI is not. Associated with *BSPlib*'s programming discipline is a simple cost model for the transmission of bulked messages. In contrast, MPI has no exposed cost model, and if one were exposed it would necessarily be based upon single messages rather than supersteps.

One consequence of the *BSPlib* philosophy is that *BSPlib* is concise and consistent, making it easy to implement efficiently. MPI's greater flexibility leads to a huge library with competing functionalities, making efficient implementation far more problematic.

The specifics of *BSPlib* were influenced by experience with the Oxford BSP library [32], the Cray SHMEM library [3], the split-phase assignments in Split-C [10], and the Green BSP library [17].

This paper presents the full definition of the C interface to *BSPlib* in Sections 2–4 (the Fortran interface is described in Ref. [24]). A quick reference table of all the 20 primitives can be found on page 35. Section 5 presents a brief description and results of applications in benchmarking, Fast Fourier Transforms, sorting, and molecular dynamics. The paper is concluded in Section 6, which discusses possible future extensions.

2. SPMD framework

Like many other communications libraries, *BSPlib* adopts an SPMD programming model. The task of writing an SPMD program will typically involve mapping a problem that manipulates a data structure of size n into p instances of a program that each manipulate an n/p sized block of the original domain. The role of *BSPlib* is to provide the infrastructure required for the *user* to take care of the data distribution, and any implied communication necessary to manipulate parts of the data structure that are on a remote process. An alternative role for *BSPlib* is to provide an architecture independent target for higher-level libraries or programming tools that automatically distribute the problem domain among the processes.

2.1. Starting and finishing SPMD code

Processes are created in a *BSPlib* program by the operations `bsp_begin` and `bsp_end`. They bracket a piece of code to be run in an SPMD manner on a number of processors. There can only be one instance of a `bsp_begin/bsp_end` within a program. If `bsp_begin` and `bsp_end` are the first and last statements in a program, then the entire *BSPlib* computation is SPMD. An alternative mode is available where a single process starts execution and determines the number of parallel processes required for the calculation. See Section 2.2 for details.

2.1.1. Syntax and parameters

```
void bsp_begin(int maxprocs);
void bsp_end(void);
```

`maxprocs` is the number of processes requested by the user.

2.1.2. Example

A trivial *BSPlib* program is shown below. The program starts as many parallel processes as there are available, each of which prints the string "Hello BSP Worldwide". The example illustrates the minimum requirements of *BSPlib* with respect to I/O. When a number of processes print a message on either standard output or standard error, the messages are multiplexed to the user's terminal in a non-deterministic manner. Therefore, this example prints the strings in an arbitrary order. All other types of I/O (e.g., user input and file access) are only guaranteed to work correctly if performed by *process zero*.

```
void main(void) {
  bsp_begin(bsp_nprocs( ));
    printf("Hello BSP Worldwide from process %d of %d\n",
      bsp_pid( ), bsp_nprocs( ));
  bsp_end( );
}
```

2.1.3. Notes

1. An implementation of *BSPlib* may spawn less than `maxprocs` processes. The actual number of processes started can be found by the enquiry function `bsp_nprocs( )`.
2. There can only be a single `bsp_begin/bsp_end` pair within a *BSPlib* program. This excludes the possibility of starting, stopping, and then restarting parallel tasks within a program, or any form of nested parallelism.
3. The process with `bsp_pid( )` $=0$ is a continuation of the thread of control that initiated `bsp_begin`. This has the effect that all the values of the local and global variables prior to `bsp_begin` are available to that process.
4. After `bsp_begin`, the environment from process zero is not inherited by any of the other processes, i.e., those with `bsp_pid( )` greater than zero. If any of them require part of zero's state, then the data must be transferred from process zero.
5. `bsp_begin` has to be the first statement of the procedure which contains the statement. Similarly, `bsp_end` has to be the last statement in the same procedure.
6. If the program is not run in a purely SPMD mode, then `bsp_init` has to be the first statement executed by the program, see the next subsection.
7. `bsp_begin(bsp_nprocs( ))` can be used to request the same number of processes as there are processors on a parallel machine.

8. All processes *must* execute bsp_end for a *BSPlib* program to complete successfully.

2.2. Simulating dynamic processes

An alternative mode of starting *BSPlib* processes is available where a single process starts execution and determines the number of parallel processes required for the calculation. The initial process can then spawn the required number of processes using bsp_begin. Execution of the spawned processes continues in an SPMD manner, until bsp_end is encountered by all the processes. At that point, all but process zero is terminated, and process zero is left to continue the execution of the rest of the program sequentially.

One problem with trying to provide this alternative mode of initialisation is that some parallel machines available today (almost all distributed memory machines, e.g. IBM SP2, Cray T3E, Parsytec GC, Hitachi SR2001) do not provide dynamic process creation. As a solution to this problem we *simulate* dynamic spawning in the following way: (1) the first statement executed by the *BSPlib* program is bsp_init which takes as its argument a name of a procedure; (2) the procedure named in bsp_init contains bsp_begin and bsp_end as its first and last statements.

2.2.1. Syntax and parameters

```
void bsp_init(void(*spmd_part)(void), int argc, char
*argv[ ])
```

spmd_part is the name of a procedure that takes no arguments and does not return a value. Its sole purpose is to isolate the SPMD part of the computation into a single procedure. The procedure will contain bsp_begin and bsp_end as its first and last statements.

2.2.2. Example

```
int nprocs; /* global variable */
void spmd_part(void) {
  bsp_begin(nprocs);
    printf("Hello BSP Worldwide from process %d of %d\n",
      bsp_pid( ),bsp_nprocs( ));
  bsp_end( );
}
void main(int argc, char *argv[ ]) {
  bsp_init(spmd_part,argc,argv);
  nprocs=ReadInteger( );
  spmd_part( );
}
```

Unlike the previous example, when the above program is executed a single process will begin execution and read a number from standard input that specifies the number of parallel processes to be spawned. The desired number of processes will then be spawned within the procedure `spmd_part`, and each process will print the string Hello BSP Worldwide.

2.3. One process stops all

The function `bsp_abort` provides a simple mechanism for raising errors in *BSPlib* programs. A single process in a potentially unique thread of control can print an error message followed by a halt of the entire *BSPlib* program. The routine is designed *not to* require a barrier synchronisation of all processes.

2.3.1. Syntax and parameters

```
void bsp_abort(char *format,...);
```

`format` is a C-style format string as used by `printf`. Any other arguments are interpreted in the same way as the variable number of arguments to `printf`.

2.3.2. Notes

1. If more than one process calls `bsp_abort` in the same superstep, then either one, all, or a subset of the processes that called `bsp_abort` may print their format string to the terminal before stopping the *BSPlib* computation.

2.4. Local enquiry functions

The *BSPlib* enquiry functions are local operations that do not require communication among the processes. They return information concerning: (1) the number of parallel processes involved in a *BSPlib* calculation; (2) a unique process identifier of the SPMD process that called the enquiry function; and (3) access to a high-precision clock.

If the function `bsp_nprocs` is called before `bsp_begin`, then it returns the number of processors which are available. If it is called after `bsp_begin` it returns p, the actual number of processes allocated to the program, where $1 \leqslant p \leqslant maxprocs$, and *maxprocs* is the number of processes requested in `bsp_begin`. Each of the p processes created by `bsp_begin` has a unique value m in the range $0 \leqslant m \leqslant p-1$. The function `bsp_pid` returns the integer m.

The function `bsp_time` provides access to a high-precision timer – the accuracy of the timer is implementation specific. The function is a local operation of each process, and can be issued at any point after `bsp_begin`. The result of the timer is the elapsed time in seconds since `bsp_begin`. The semantics of `bsp_time` is as though there were p timers, one per process. *BSPlib* does *not impose any synchronisation requirements between the timers on different processes.*

2.4.1. Syntax and parameters

```
int bsp_nprocs(void);
int bsp_pid(void);
double bsp_time(void);
```

2.5. Superstep

A *BSPlib* calculation consists of a sequence of supersteps. During a superstep each process can perform a number of computations on data held locally at the start of the superstep and may communicate data to other processes. Any communications within a superstep are guaranteed to occur by the end of the superstep, where all processes synchronise at a barrier – *BSPlib* has no form of subset synchronisation.

The end of one superstep and the start of the next is identified by a call to the library procedure `bsp_sync`. Communication initiated during a superstep is *not guaranteed* to occur until `bsp_sync` is executed; this is even the case for the unbuffered variants of communication.

2.5.1. Syntax and parameters

```
void bsp_sync(void);
```

3. Direct remote memory access

One way of performing data communication in the BSP model is to use DRMA communication facilities. Some parallel programming libraries require that the data structures used in DRMA operations have to be held at statically allocated memory locations. *BSPlib* does not have this restriction, which enables communication in certain heterogeneous environments, and allows communication into any type of contiguous data structure including stack or heap allocated data. This is achieved by allowing a process to manipulate certain *registered* areas of a remote memory which have been previously made available by the corresponding processes. In this registration procedure, processes use the operation `bsp_push_reg` to announce the address of the start of a local area which is available for global remote use.

The operation `bsp_put` deposits locally held data into a registered remote memory area on a target process, without the active participation of the target process. The operation `bsp_get` reaches into the registered local memory of another process to copy data values held there into a data structure in its own local memory.

Allowing a process to arbitrarily manipulate the memory of another process, without the involvement of that process, is potentially dangerous. The mechanisms we propose here exhibit different degrees of *safety* depending on the buffering requirements of the communication operations. The right choice of buffering depends on the class of applications and the desired goals, and has to be made by the user.

There are four forms of buffering with respect to the DRMA operations:

Buffered on destination: Writing data into registered areas will happen *at* the end of the superstep, once all *remote reads* have been performed.

Unbuffered on destination: Data communication into registered areas can take effect at any time during the superstep. Therefore, for safety, no process should change the destination data structures used during the course of the superstep.

Buffered on source: If the source data structure is in the memory of the process that issues a communication action (i.e., a put), then a copy of the data is made at the time the communication action is issued; the source data structure can therefore be changed by the user immediately after communications are issued. Alternatively, if the source data structure is on a remote process (i.e., a get), then the data is read on the remote process at the end of the superstep, *before any remote writes are performed.*

Unbuffered on source: The data transfer resulting from a call to a communication operation may occur at any time between the time of issue and the end of the superstep.Therefore, for safety, no process should change the source data structures used during the course of the superstep.

The various buffering choices are crucial in determining the safety of the communication operation, i.e., the conditions which guarantee correct data delivery as well as its effects on the processes involved in the operation. However, it should be noted that even the most cautious choice of buffering mode does not completely remove non-determinism. For example, if more than one process transfers data into overlapping memory locations, then the result at the overlapping region will be non-deterministically chosen; it is implementation dependent which one of the many "colliding" communications should be written into the remote memory area.

3.1. Registration

A *BSPlib* program consists of p processes, each with its own local memory. The SPMD structure of such a program produces p local instances of the various data structures used in the program. Although these p instances share the same name, they will not, in general, have the same physical address. Due to stack or heap allocation, or due to implementation on a heterogeneous architecture, one might find that the p instances of variable x have been allocated at up to p different addresses.

To allow *BSPlib* programs to execute correctly we require a mechanism for relating these various addresses by creating associations called *registrations.* A registration is created when each process calls `bsp_push_reg` and, respectively, provides the address and the extent of a local area of memory. Both types of information are relevant as processes can create new registrations by providing the same addresses, but different extents. The semantics adopted for registration enables procedures called within supersteps to be written in a modular way by allowing newer registrations to temporarily replace older ones. However, the scheme adopted does not impose the strict nesting of push–pop pairs that is normally associated with a stack. This provides the benefits of encapsulation provided by a stack, whilst providing the flexibility associated with a heap-based discipline. In line with superstep semantics, *registration takes effect at the next barrier synchronisation.*

A registration association is destroyed when each process calls `bsp_pop_reg` and provides the address of its local area participating in that registration. A runtime error will be raised if these addresses (i.e., one address per process) do not refer to the same registration association. In line with superstep semantics, *de-registration takes effect at the next barrier synchronisation.*

One interpretation of the registration mechanism is that there is a sequence of registration slots that are accessible by all the processes. If each process i executes

$$\texttt{bsp\_push\_reg}(indent_i, size_i)$$

then the entry $\langle\langle ident_0, size_0\rangle, \ldots, \langle ident_{p-1}, size_{p-1}\rangle\rangle$ is added to the front of the sequence of registration slots. The intent of registration is to make it simple to refer to remote storage areas without requiring their locations to be explicitly known. A reference to a registered area in a `bsp_put` or `bsp_get` is translated to the address of the corresponding remote area in its most recent registration slot. For example, if tgt_l is used in a put executed on process l,

$$\texttt{bsp\_put}(r, src, tgt_l, offset, nbytes)$$

and the registration sequence [1] is $+\!\!+\ ss +\!\!+ \overline{ss}$, where entry s is the most recent entry containing tgt_l (i.e., the lth element of s is $\langle tgt_l, n_l\rangle$, and there is no entry $\bar{s}$ in ss such that the lth element of $\bar{s}$ is $\langle tgt_l, m_l\rangle$), then the effect is to transfer *nbytes* of data from the data structure starting at address *src* on process l into the contiguous memory locations starting at $tgt_r + offset$ on process r, where the base address tgt_r comes from the same registration slot s as tgt_l. Rudimentary bounds checking may be performed on the communication, such that a runtime error can be raised if $offset + nbytes > n_r$.

The effect of the de-registration

$$\texttt{bsp\_pop\_reg}(ident_l)$$

is that given the registration sequence

$$ss +\!\!+ [\langle\langle ident_0, size_0\rangle, \ldots, \langle ident_{p-1}, size_{p-1}\rangle +\!\!+ \overline{ss},$$

and suppose that there does not exist an entry $\bar{s}$ in ss such that the lth element of $\bar{s}$ is $\langle ident_l, m_l\rangle$, then the registration sequence is changed to $ss +\!\!+ \overline{ss}$ *at the start of the next superstep.* A runtime error will be raised if differing processes attempt to de-register a different registration slot during the same de-registration. For example, if process p_0 registers x twice, and process p_1 registers x followed by y, then a runtime error will be raised if both processes attempt to de-register x. This error is due to the active registration for x referring to a different registration slot on each process.

[1] The operator $+\!\!+$, is used to concatenate two sequences together.

3.1.1. Syntax and parameters

```
void bsp_push_reg(const void *ident, int size);
void bsp_pop_reg(const void *ident);
```

`ident` is a previously initialised variable denoting the address of the local area being registered or de-registered.

`size` is a non-negative integer denoting the extent, in bytes, of the area being registered for use in bounds checking within the library.

3.1.2. Notes

1. `bsp_push_reg` takes effect at the end of the superstep. DRMA operations may use the registered areas from the start of the next superstep.
2. DRMA operations are allowed to use memory areas that have been de-registered in the same superstep, as `bsp_pop_reg` only takes effect at the end of a superstep.
3. Communication into unregistered memory areas raises a runtime error.
4. Registration is a property of an area of memory and not a reference to the memory. There can therefore be many references (i.e., pointers) to a registered memory area.
5. If only a subset of the processes are required to register data because a program may have no concept of a *commonly named* memory area on all processes, then all processes must call `bsp_push_reg` although some may register the memory area `NULL`. This memory area is regarded as unregistered.
6. While registration is designed for "full duplex" communication, a process can do half duplex communication by, appropriately, registering an area of size 0.
7. It is an error to provide negative values for the size of the registration area.
8. Since on each process static data structures are allocated at the same address (this is not always the case, as some optimising C compilers *un-static* statics), the registration slot in such cases will have the form:

$$\underbrace{\langle\langle ident_0, n_0\rangle, \ldots, \langle ident_0, n_{p-1}\rangle\rangle}_{p \text{ copies}}$$

Even though static data structures are allocated at the same address, they still have to be registered.

3.2. Copy to remote memory

The aim of `bsp_put` and `bsp_hpput` is to provide an operation akin to `memcpy` available in the Unix `<string.h>` library. Both operations copy a specified number of bytes, from a byte addressed data structure in the local memory of one process into contiguous memory locations in the local memory of another process.

The distinguishing factor between these operations is provided by the buffering choice.

The semantics *buffered on source, buffered on destination* is used for `bsp_put` communications. While the semantics is clean and safety is maximised, puts may unduly tax the memory resources of a system. Consequently, *BSPlib* also provides a *high performance put* operation `bsp_hpput` whose semantics is *unbuffered on source, unbuffered on destination*. The use of this operation requires care as correct data delivery is only guaranteed if: (1) no communications alter the source area; (2) no subsequent local computations alter the source area; (3) no other communications alter the destination area; and (4) no computation on the remote process alters the destination area during the entire superstep. The main advantage of this operation is its economical use of memory. It is therefore particularly useful for applications which repeatedly transfer large data sets.

3.2.1. Syntax and parameters

```
void bsp_put(
    int pid, const void *src,
    void *dst, int offset, int nbytes);
void bsp_hpput(
    int pid, const void *src,
    void *dst, int offset, int nbytes);
```

`pid` is the identifier of the process where data is to be stored.

`src` is the location of the first byte to be transferred by the put operation.

The calculation of `src` is performed on the process that initiates the put.

`dst` is the location of the first byte where data is to be stored. It must be a previously registered area.

`offset` is the displacement in bytes from `dst` where `src` will start copying into. The calculation of `offset` is performed by the process that initiates the put.

`nbytes` is the number of bytes to be transferred from `src` into `dst`.It is assumed that src and dst are addresses of data structures that are at least `nbytes` in size. The data communicated can be of arbitrary size. It is *not required* to have a size which is a multiple of the word size of the machine.

3.2.2. Example

The `reverse` function shown below highlights the interaction between registration and put communications. This example defines a simple collective communication operation, in which all processes have to call the function within the same superstep. The result of the function on process i will be the value of the parameter `x` from process `bsp_nprocs( )` $- i - 1$.

```
int reverse(int x) {
  bsp_push_reg(&x, sizeof(int));
  bsp_sync( );
```

```
    bsp_put(bsp_nprocs( )-bsp_pid( )-1,&x,&x,0,sizeof(int));
    bsp_sync( );
    bsp_pop_reg(&x);
    return x;
  }
```

By the end of the first superstep, identified by the first `bsp_sync`, all the processes will have registered the parameter `x` as being available for remote access by any subsequent DRMA operation. During the second superstep, each process transfers its local copy of the variable `x` into a remote copy on process `bsp_nprocs( ) − bsp_pid( ) − 1`. Although communications occur to and from the same variable within the same superstep, the algorithm does not suffer from problems of concurrent assignment because of the buffered on source, buffered on destination semantics of `bsp_put`. This buffering ensures conflict-free communication between the outgoing communication from `x`, and any incoming transfers from remote processes.The de-register at the end of the function reinstates the registration properties that were active on entry to the function *at the next bsp_sync encountered during execution.*

3.2.3. Example

The procedure `put_array` shown below has a semantics defined by the concurrent assignment:

$$\forall i \in \{0,\ldots,n-1\} \quad xs[xs[i]] := xs[i]$$

Conceptually, the algorithm manipulates a global array *xs* of *n* elements that are distributed among the processes. The role of *BSPlib* is to provide the infrastructure for the user to take care of the data distribution, and any implied communication necessary to manipulate parts of the data structure that are on a remote process. Therefore, if the user distributes the global array in a block-wise manner (i.e., process zero gets elements 0 to *n*/*p* − *1*, process one gets *n*/*p* to *2n*/*p* − *1*, etc.) with each process owning an *n*/*p* chunk of elements, then the *BSPlib* communications necessary to perform the concurrent assignment are shown below.

```
  void put_array(int *xs, int n) {
    int i,dst_pid,dst_idx,p=bsp_nprocs( ),n_over_p=n/p;
    if ((n % p) !=0
      bsp_abort("{put_array}    n=%d    not    divisible    by
      p=%d",n,p);
    bsp_push_reg(xs,n_over_p*sizeof(int));
    bsp_sync( );
    for(i=0;i<n_over_p;i++){
      dst_pid=xs[i]/n_over_p;
      dst_idx=xs[i]%n_over_p;
      bsp_put(dst_pid,&xs[i],xs,dst_idx*sizeof(int),size-
      of(int));
    }
    bsp_sync( );
    bsp_pop_reg(xs);
  }
```

The procedure highlights the use of `bsp_abort` and the offset parameter in `bsp_put`. Each process's local section of the array `xs` is registered in the first superstep. Next, *n/p* puts are performed, in which the global numbering used in the distributed array (i.e., indices in the range 0 through $n - 1$), are converted into pairs of process identifier and local numbering in the range 0 to $n/p - 1$. Once the conversion from the global scheme to process-id/local index has been performed, elements of the array can be transferred into the correct index on a remote process. It should be noted that if the value of the variable `dst_pid` is the same as `bsp_pid( )`, then a local assignment (i.e., memory copy) will occur *at the end of the superstep*. In this example, buffering is necessary as processes need to read data before it is overwritten.

3.2.4. Notes

1. The destination memory area used in a put has to be registered. It is an error to communicate into a data structure that has not been registered.
2. The source of a put does *not have to be registered.*
3. If the destination memory area `dst` is registered with size x, then it is a bounds error to perform the communication $\texttt{bsp\_put}(\texttt{pid}, \texttt{src}, \texttt{dst}, o, n)$ if $o + n > x$.
4. A communication of zero bytes does nothing.
5. A process can communicate into its own memory if $\texttt{pid} = \texttt{bsp\_pid( )}$. However, for `bsp_put`, due to the *buffered at destination* semantics, the memory copy only takes effect *at the end of the superstep.*
6. The process numbering and offset parameter start from zero.

3.3. Copy from remote memory

The `bsp_get` and `bsp_hpget` operations reach into the local memory of another process and copy previously registered remote data held there into a data structure in the local memory of the process that initiated them.

The semantics *buffered on source*, *buffered on destination* is used for `bsp_get` communications. This semantics means that the value taken from the source on the remote process by the get, is the value available once the remote process finishes executing all its superstep computations. Furthermore, writing the value from the remote process into the destination memory area on the initiating process only takes effect at the end of the superstep after all remote reads from any other bsp_get operations are performed, *but before* any data is written by any bsp_put. Therefore, computation and buffered communication operations within a superstep can be *thought* to occur in the following order:

1. local computation is performed; also, when a `bsp_put` is excecuted, the associated source data is read;

2. the source data associated with all bsp_gets are read;
3. data associated with any bsp_put or bsp_get are written into the destination data structures.

A high-performance version of get, bsp_hpget, provides the *unbuffered on source, unbuffered on destination* semantics in which the two-way communication can take effect at any time during the superstep.

3.3.1. Syntax and parameters

```
void bsp_get(
    int pid, const void *src, int offset,
    void *dst, int nbytes);
void bsp_hpget(
    int pid, const void *src, int offset,
    void *dst, int nbytes);
```

pid is the identifier of the process where data is to be obtained from.

src is the location of the first byte from where data will be obtained. src must be a previously registered memory area.

offset is an offset from src where the data will be taken from. The calculation of offset is performed by the process that initiates the get.

dst is the location of the first byte where the data obtained is to be placed. The calculation of dst is performed by the process that initiates the get.

nbytes is the number of bytes to be transferred from src into dst. It is assumed that src and dst are addresses of memory areas that are at least nbytes in size.

3.3.2. Example

The function bsp_sum defined below is a collective communication (i.e., all processes have to call the function), such that when process *i* calls the function with an array xs containing $\texttt{nelem}_i$ elements, then the result on *all* the processes will be the sum of all the arrays from all the processes.

```
int bsp_sum(int*xs, intnelem){
  int *local_sums, i, j, result = 0, p = bsp_nprocs( );
  for(j = 0; j<nelem; j++) result += xs[j];
  bsp_push_reg(&result, sizeof(int));
  bsp_sync( );
  local_sums = calloc(p, sizeof(int));
  if (local_sums = = NULL)
    bsp_abort("{bsp_sum} no memory for %d int", p);
  for(i = 0; i<p; i++)
    bsp_hpget(i, &result, 0, &local_sums[i], sizeof(int));
  bsp_sync( );
  result = 0;
```

```
    for(i = 0; i<p; i++) result += local_sums[i];
    bsp_pop_reg(&result);
    free(local_sums);
    return result;
  }
```

The function contains three supersteps. In the first, the local array `xs` of each process is summed and assigned to the variable `result`. This variable is then registered for communication in the subsequent superstep. Next, each local `result` is broadcast into the `bsp_pid( )`th element of `local_sums` on every process. Unlike the previous examples, an unbuffered communication is used in preference to a buffered `bsp_get` because the variable `result` is not used in any local computation during the same superstep as the communication. In the final superstep of the algorithm, each process returns the sum of the p values obtained from each process.

3.3.3. Notes

1. The source memory area used in a get has to be registered. It is an error to fetch from a data structure that has not been registered.
2. The destination of a get does *not have to be registered.*
3. If the source memory area `src` is registered with size x, then it is a bounds error to perform the communication `bsp_get(pid, src,` o`, dst,` n`)` if $o + n > x$.
4. A communication of zero bytes does nothing.
5. A process can read from its own memory if `pid` = `bsp_pid( )`. However, due to the *buffered at destination* semantics of `bsp_get`, the memory copy only takes effect *at the end of the superstep*; i.e, the source data is read and then written at the end of the superstep.

4. Bulk synchronous message passing

DRMA is a convenient style of programming for BSP computations which can be statically analysed in a straightforward way. It is less convenient for computations where the volumes of data being communicated in supersteps are irregular and data dependent, and where the computation to be performed in a superstep depends on the quantity and form of data received at the start of that superstep. A more appropriate style of programming in such cases is BSMP.

In BSMP, a non-blocking send operation is provided that delivers messages to a system buffer associated with the destination process. The message is guaranteed to be in the destination buffer at the beginning of the subsequent superstep, and can be accessed by the destination process only during that superstep. If the message is not accessed during that superstep it is removed from the buffer. In keeping with BSP superstep semantics, the messages sent to a process during a superstep have no implied ordering at the receiving end; a destination buffer may therefore be viewed as a queue, where the incoming messages are enqueued in arbitrary order and are dequeued (accessed) in that same order. Note that although messages are typically identified with tags, *BSPlib* provides no tag-matching facility for the out-of-order access of specific incoming messages.

In *BSPlib*, bulk synchronous message passing is based on the idea of two-part messages, a fixed-length part carrying tagging information that will help the receiver to interpret the message, and a variable-length part containing the main data payload. We will call the fixed-length portion the *tag* and the variable-length portion the *payload*. The length of the tag is required to be fixed during any particular superstep, but can vary between supersteps. The buffering mode of the BSMP operations is *buffered on source, buffered on destination*. We note that this buffering classification is a semantic description; it does not necessarily describe the underlying implementation.

4.1. Choose tag size

Allowing the user to set the tag size enables the use of tags that are appropriate for the communication requirements of each superstep. This should be particularly useful in the development of subroutines either in user programs or in libraries.

The procedure must be called collectively by all processes. A change in tag size takes effect in the following superstep; the tag size then becomes *valid*.

4.1.1. Syntax and parameters

```
void bsp_set_tagsize (int *tag_nbytes);
```

`tag_nbytes` on entry to the procedure, specifies the size of the fixed-length portion of every message in the subsequent supersteps; the default tag size is zero. On return from the procedure, `tag_nbytes` is changed to reflect the *previous* value of the tag size. This can be used to reinstate the previous state of the system.

4.1.2. Notes

1. The tag size of outgoing messages is prescribed by the tag size that is valid in the current superstep.
2. The tag size of messages in the system queue is prescribed by the tag size that was valid in the previous superstep.
3. `bsp_set_tagsize` must be called by *all* processes with the same argument in the same superstep. In this respect, it is similar to `bsp_push_reg`.
4. `bsp_set_tagsize` takes effect in the next superstep.
5. Given a sequence of `bsp_set_tagsize` within the same superstep, the value of the last of these will be used as the tag size for the next superstep.
6. The default tag size is 0.

4.2. Send to remote queue

The `bsp_send` operation is used to send a message that consists of a tag and a payload to a specified destination process. The destination process will be able to access the message during the subsequent superstep. The bsp_send operation copies both the tag and the payload of the message before returning. The tag and payload variables can therefore be changed by the user immediately after the bsp_send. Messages sent by bsp_send are *not* guaranteed to be received in any particular order by the destination process. This is the case even for successive calls of bsp_send from one process with the same value for pid.

4.2.1. Syntax and parameters

```
void bsp_send(
    int pid, const void *tag,
    const void *payload, int payload_nbytes);
```

pid is the identifier of the process where data is to be sent.

tag is a token that can be used to identify the message. Its size is determined by the value specified in bsp_set_tagsize.

payload is the location of the first byte of the payload to be communicated.

payload_nbytes is the size of the payload.

4.2.2. Notes

1. The size of the tag used in bsp_send will depend on either the size of tag that was valid in the previous superstep, or the size specified by the last bsp_set_-tagsize *issued in the previous superstep.*
2. If the payload size is zero, then a message that only contains the tag will be sent. Similarly, if the tag size is zero, then a message just containing the payload will be sent. If both the tag and payload are zero, a message that contains neither tag nor payload *will be sent*.
3. If the tag size is zero, then the tag argument may be NULL. Similarly, if the payload size is zero, then the payload argument may be NULL.

4.3. Number of messages in queue

The function bsp_qsize is an enquiry function that returns the number of messages that were sent to this process in the previous superstep and have not yet been consumed by a bsp_move. Before any message is consumed by bsp_move, the total number of messages received will match those sent by any bsp_send operations in the previous superstep. The function also returns the accumulated size of all the payloads of the unconsumed messages. This operation is intended to help the user to allocate an appropriately sized data structure to hold all the messages that were sent to a process during a superstep.

4.3.1. Syntax and parameters

```
void bsp_qsize(int *nmessages, int *accum_nbytes);
```

`nmessages` becomes the number of messages sent to this process in the previous superstep by using `bsp_send`.

`accum_nbytes` is the accumulated size of all the message payloads sent to this process.

4.3.2. Notes

1. `bsp_qsize` returns the number of messages and their accumulated size in the system queue at the point the operation is called; the number returned therefore decreases after any `bsp_move` operation.

4.4. Getting the tag of a message

To receive a message, the user should use the procedures `bsp_get_tag` and `bsp_move`. The operation `bsp_get_tag` returns the tag of the first message in the queue. The size of the tag will depend on the value set by `bsp_set_tagsize`.

4.4.1. Syntax and parameters

```
void bsp_get_tag(int *status, void *tag)
```

`status` becomes −1 if the system queue is empty. Otherwise it becomes the length of the payload of the first message in the queue. This length can be used to allocate an appropriately sized data structure for copying the payload using `bsp_move`.

`tag` is unchanged if the system queue is empty. Otherwise it is assigned the tag of the first message in the queue.

4.5. Move from queue

The operation `bsp_move` copies the payload of the first message in the system queue into `payload`, and removes that message from the queue.

Note that `bsp_move` serves to flush the corresponding message from the queue, while `bsp_get_tag` does not.This allows a program to get the tag of a message (as well as the payload size in bytes) before obtaining the payload of the message. It does, however, require that even if a program only uses the fixed-length tag of incoming messages the program must call `bsp_move` to get successive message tags.

4.5.1. Syntax and parameters

```
void bsp_move(void *payload, int reception_nbytes);
```

`payload` is an address to which the message payload will be copied. The system will then advance to the next message.

`reception_nbytes` specifies the size of the reception area where the `payload` will be copied into. At most `reception_nbytes` will be copied into `payload`.

4.5.2. Example

In the algorithm shown below, an *n* element vector distributed into *n*/*p* chunks on *p* processes undergoes a communication whereby all the non-zero elements from all the *p* chunks are broadcast to *all* the processes. Due to the sparse nature of the problem, the communication pattern is well suited to BSMP as the amount and placement of data is highly data dependent.

```
int all_gather_sparse_vec(float *dense, int n_over_p,
                                        float **sparse_out,
                                        int  **sparse_ivec_-
  out){
  int global_idx, i, j, tag_size, p = bsp_nprocs( ),
    nonzeros, nonzeros_size, status, *sparse_ivec;
  float *sparse;
  tag_size = sizeof(int);
  bsp_set_tagsize(&tag_size);
  bsp_sync( );
  for(i = 0; i<n_over_p; i++)
    if (dense[i]! = 0.0) {
         global_idx = (n_over_p*bsp_pid( ))+i;
         for(j = 0; j<p; j++)
     bsp_send(j, &global_idx, &dense[i], sizeof(float));
      }
  bsp_sync( );
  bsp_qsize (&nonzeros, &nonzeros_size);
  if (nonzeros>0) {
    sparse = calloc(nonzeros, sizeof(float));
    sparse_ivec = calloc(nonzeros, sizeof(int));
    if (sparse = = NULL || sparse_ivec = = NULL)
       bsp_abort("Unable to allocate memory");
    for(i = 0; i<nonzeros; i++){
       bsp_get_tag(&status, &sparse_ivec[i]);
       if (status! = sizeof(float))
     bsp_abort("Should never get here");
       bsp_move(&sparse[i], sizeof(float));
      }
  }
  bsp_set_tagsize(&tag_size);
  *sparse_out = sparse;
  *sparse_ivec_out = sparse_ivec;
  return nonzeros;
}
```

The algorithm contains three supersteps. In the first superstep, the tag size of the messages in the subsequent supersteps is set to the size of an integer. The size of the tag prior to the `bsp_set_tagsize` is remembered so that it can be reinstated at the end of the procedure. Next, the non-zero elements of the vector are broadcast to each process using `bsp_send`. The tag for each send operation is set to be the position of the vector element within the global array of n elements; the payload of the message will be the non-zero element. A `bsp_sync` is used to ensure that all the `bsp_send` operations are delivered to the system queue on the remote processes, and then `bsp_qsize` is used to determine how many messages arrived at each process. This information is used to allocate a pair of arrays (one for array indices, and one for values), which have the messages copied into them by a `bsp_move` operation.

4.5.3. Notes

1. The payload length is always measured in bytes.
2. `bsp_get_tag` can be called repeatedly and will always copy out the same tag until a call to `bsp_move`.
3. If the payload to be received is larger than `reception_nbytes`, the payload will be truncated.
4. If `reception_nbytes` is zero this simply "removes" the message from the system queue.This should be efficient in any implementation of the library.

4.6. A lean method for receiving a message

The operation `bsp_hpmove` is a non-copying method of receiving messages that is available in languages with pointers such as C.

We note that since messages are referenced directly they must be properly aligned and contiguous. This puts additional requirements on the library implementation that would not be there without this feature, as it requires the availability of sufficient contiguous memory. The storage referenced by these pointers remains valid until the end of the current superstep.

4.6.1. Syntax and parameters

```
int bsp_hpmove(void **tag_ptr, void **payload_ptr);
```

`bsp_hpmove` is a function which returns -1 if the system queue is empty. Otherwise it returns the length of the payload of the first message in the queue and: (1) places a pointer to the tag in `tag_ptr`; (2) places a pointer to the payload in `payload_ptr`; and (3) removes the message (by advancing a pointer representing the head of the queue).

5. Applications

BSPlib has been developed hand in hand with a number of applications. The design of *BSPlib* is based on the theory of the BSP model, but the library has been tested in applications and it was further refined based on practical experience. The primitives of *BSPlib* have all been found useful in applications. Together, the primitives form a complete set. In this section, we present four applications: benchmarking, Fast Fourier Transform, sorting, and molecular dynamics. The first two use the DRMA approach, while the others use BSMP.

5.1. Benchmarking

One of the strengths of the BSP model is the ability to accurately predict the cost of parallel algorithms [21,33]. This is achieved by constructing analytical formulae that are parameterised by three constants capturing the computation, communication, and synchronisation performance of a parallel machine. In this subsection, a series of synthetic benchmarks are described which empirically calculate these constants.

5.1.1. BSP cost analysis

The superstep structure of BSP programs facilitates cost analysis because the barrier synchronisation that delimits a superstep ensures that the cost of a sequence of supersteps is simply the sum of the costs of the separate supersteps. As a single superstep can be decomposed into three distinct phases of local computation, communication, and barrier synchronisation, it is natural to express the cost of a superstep by formulae that have the structure:

$$\text{cost of a superstep} = \underset{\text{processes}}{\mathbf{MAX}}\, w_i + \underset{\text{processes}}{\mathbf{MAX}}\, h_i\, g + l,$$

where i ranges over the processes. Intuitively, the cost of a superstep is the cost of the process that performs the largest local computation (i.e., $\mathbf{MAX}\, w_i$), added to the cost of the process that performs the largest communication ($\mathbf{MAX}\, h_i g$), added to a constant cost l that arises from the barrier synchronisation. Communication costs are based on the observation that the process that has the largest amount of data entering or leaving will form the bottle-neck in the system. The global size of the communication phase is expressed by $h = \mathbf{MAX}\, h_i$, and the phase is called an h-relation. Multiplying the maximum number of words h by g, the communication cost per word, gives the cost in the same units as for the computation.

To make the costs meaningful, and to allow comparisons between different parallel computers, we express the costs in flop time units, where one unit is the time it takes to perform one floating point operation (flop) on the target architecture. Therefore, we are interested in benchmarking the following three *architecture-dependent* BSP machine parameters:

s is the speed of computation of a process in flop/s (i.e., the number of floating point operations per second). It is used to calibrate g and l.

g is the cost in flop time units to communicate a single word to a remote process, *under the conditions where all processors are simultaneously communicating*.

l is the synchronisation latency cost in flop time units. It is the amount of time needed for all processors to synchronise.

5.1.2. Calculating the BSP machine parameters

Values of the BSP cost parameters that were calculated using the Oxford implementation of *BSPlib* [25] are shown in Table 1. [2] The machines are ordered by decreasing computing rate s. Note that a low value of the dimensionless parameters g and l may either mean that the machine has a powerful communication network, *or* that it is has a particularly poor computing rate. The motivation for expressing g and l in flop time units is that it gives a measure of the "resource balance" of the system, which is important to the algorithm designer.

The computing rate s depends heavily on the kind of computations being done; to express the rate in a single value, we use the average of the following two measured rates: (1) The rate for an inner product computation that is mostly out of cache; this serves as a lower bound on the achievable rate. (2) The rate for a dense matrix multiplication that is mostly in cache; this serves as an upper bound. Note that taking the average yields a value of s that is far lower than the peak computing rate.

The BSP parameter l is obtained by timing a mid-stream sample of a repeated number of barrier synchronisations.

The BSP parameter g is obtained as follows. It is clear from the BSP cost formula that a good strategy for writing efficient BSP programs is to balance communication between processes; this is because h is a *maximum* over the processes. Therefore, we measure g by using balanced communications. In our benchmark, we use two communication patterns. The first is a *localised* communication pattern that performs a cyclic shift of n 32-bit words between neighbouring processors using the `bsp_hpput` operation. This is an n-relation. As the expected cost of this benchmark is $ng + l$, we can obtain the value of g from the measured time t in seconds by $g = (ts - l)/n$, where n is chosen sufficiently large. Note that g is an asymptotic value. As g represents the cost of communication when all processors are simultaneously communicating, this benchmark provides a lower bound on g because each processor only injects one message into the communication network.

The second pattern may be called *global*, since it is a total exchange where each processor sends a message of $n/(p-1)$ 32-bit words to each of the $p-1$ other processors.This is also an n-relation; g can be obtained in the same way as before.This benchmark injects the maximum of $p(p-1)$ messages into the network. Parallel computers have greater difficulty in achieving scalable communication for patterns of communication that move lots of data to many destinations. As no scalable architecture can provide $O(p^2)$ dedicated wires – this would be too expensive – sparser interconnection networks are used in practice. For example, the Cray T3D and T3E use a 3D torus, while the IBM SP2 uses a hierarchy of 8-node fully connected crossbar switches. This will be reflected in increasing communication costs. The value of g for a total exchange therefore provides a good upper bound on g.

It is important to note that the lower and upper bounds for g are still quite close for most machines, even though they represent very different communication pat-

[2] A more detailed version of this table is continuously updated at `http://www.bsp-worldwide.org/implmnts/oxtool/params.html`.

Table 1
The BSP cost parameters for a variety of shared memory and distributed memory parallel machines

Machine	s (Mflop/s)	p	l	g (local)	g (global)
SGI Origin 2000	101	2	804	7.0	8.3
		4	1789	9.1	10.2
		8	3914	13.2	15.1
		16	15961	38.6	44.9
SGI PowerChallenge	74	2	1132	9.8	10.2
		4	1902	9.8	9.3
Pentium Pro NOW	61	2	52745	486.3	484.5
(10 Mbit/s		4	139981	1098.7	1128.5
shared Ethernet)		8	539159	2171.8	1994.1
		16	2884273	3708.2	3614.6
Cray T3E	47	2	269	0.9	2.6
		4	357	0.9	2.1
		8	506	0.8	1.6
		16	751	1.0	1.7
		32	1252	1.3	1.9
IBM SP2	26	2	1903	6.3	7.8
		4	3583	6.4	8.0
		8	5412	6.9	11.4
Cray T3D	12	2	164	0.7	1.0
		4	168	0.7	0.8
		8	175	0.8	0.8
		16	181	0.9	1.0
		32	201	1.1	1.4
		64	148	1.0	1.7
		128	301	1.1	1.8
		256	387	1.2	2.4
Sparc-20 SMP	10	2	54	3.0	3.4
		4	118	3.3	4.1

The computing rate is for single-precision `float` operations and the communication rate for 32-bit words.

terns. This is a posteriori justification for basing communication costs exclusively on a count of h, and not on more detailed knowledge of the communication pattern. When modelling the cost of algorithms, it is advisable to err on the safe side and use the upper bound value of g. For a more detailed discussion on benchmarking BSP parameters, and a description of the techniques that were used in the implementation of *BSPlib* to minimise the variance in g, see Refs. [25,38,12,22].

5.2. Fast Fourier transform

The Fast Fourier Transform (FFT) is important in many areas of scientific computation.A bulk synchronous parallel implementation of the FFT is part of BSPPACK, a package of parallel numerical software that is being developed at

Utrecht University. At present, BSPPACK contains programs for dense LU decomposition [5], FFT, sparse matrix–vector multiplication [6], sparse Cholesky factorisation [4], as well as a program for BSP benchmarking of parallel computers. BSPPACK is written in C, with *BSPlib* used for parallelism. The main goal of BSPPACK is to teach how to use the BSP model in numerical applications, and to serve as a prototype for optimised numerical software. The following is a brief description of the FFT program of BSPPACK, which implements a radix-2 algorithm [40].

A radix-2 FFT performs a sequence of operations on a complex vector of length n, where n is a power of two. In each stage of the computation, vector components are modified in pairs, each input pair yielding a new output pair. In stage $k, 0 \leqslant k < \log_2 n$, the components of a pair are at distance 2^k. Assume that we have p processors, where p is a power of 2. The basic idea of the parallel FFT is to permute the vector such that during the next $\log_2 n - \log_2 p$ stages both components of the pairs are on the same processor. Papadimitriou and Yannakakis [35] observed that this can be done. The basic idea was first analysed in a BSP context by Valiant [39] and it was incorporated in a BSP algorithm by McColl [31]. For the common case $p \leqslant \sqrt{n}$, this parallel FFT requires only one permutation, which costs $ng/p + l$. (At the start, no permutation is needed, provided the input vector is already suitably distributed. A permutation at the end can be avoided if we accept the output vector in its current distribution.) This particular case has been implemented by Culler et al. [11] within the framework of the LogP model.

The piece of code below illustrates the use of *BSPlib* in the permutation function of the FFT program. The communication pattern is entirely regular, so that it is natural to use DRMA. Note that the automatic buffering of `bsp_put` makes it unnecessary to use a temporary array for storing the new vector. The vector `x` is registered and de-registered outside the function, because this has to be done only once, whereas the function may be invoked several times.

```
void bsppermute(complex_t *x, int n){
  /* This function permutes the vector x of length n,
    where x is distributed by the block distribution.
    It is assumed that x has already been registered. */
  int j, sigma, dst_pid, dst_idx, p=bsp_nprocs( ), n_over_-
  p=n/p;
  for (j=0; j<n_over_p; j++){
    sigma=j*p + bsp_pid( );
    dst_pid=sigma / n_over_p;
    dst_idx=sigma % n_over_p;
    bsp_put(dst_pid, & x[j], x, dst_idx*sizeof(complex_t),
     sizeof(complex_t));
  }
bsp_sync( );
}
```

Fig. 1 presents the absolute speedup of an FFT of length 16 384 on a Cray T3E with 32 processors, using version 1.1 of the Oxford BSP Toolset implementation of *BSPlib*. The speedup is obtained by comparing to a sequential program with a similar level of optimisation, which runs at 10.9 Mflop/s. (We call the speedup of a parallel program *absolute* if it is the speedup compared to a good sequential program; we call it *relative* if it is obtained by comparing to the same parallel program run with $p = 1$.) Note that the measured speedup is nearly ideal, but that this is flattered by cache effects; the superlinear speedup of 4.4 on four processors shows that the local problems fit better into the caches of the processors.

5.3. Randomised sample sort

One approach for parallel sorting that is suitable for BSP computing is randomised sample sort. The sequential predecessor to the algorithm is sequential samplesort [14], proposed by Frazer and McKellar as a refinement of Hoare's quicksort [26]. Samplesort uses a random sample set of input keys to select splitters, resulting in greater balance – and therefore a lower number of expected comparisons – than quicksort. The fact that the sampling approach could be useful for splitting keys in a balanced manner over a number of processors was discussed in the work of Huang and Chow [27] and Reif and Valiant [36]. Its use was analyzed in a BSP context by Gerbessiotis and Valiant [15].

The basic idea behind randomised sample sort in a p-processor system is the following.

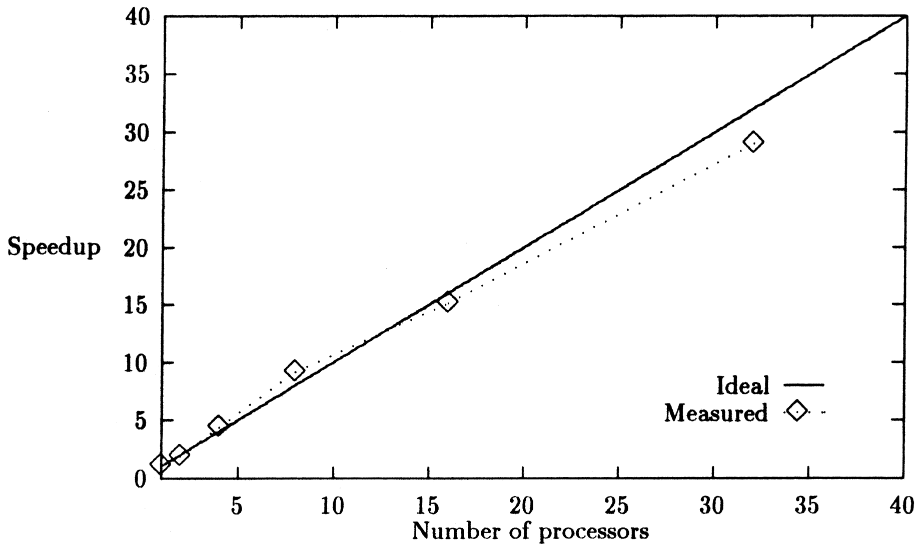

Fig. 1. Absolute speedup of parallel FFT on a Cray T3E with 32 processors. The problem size is $n = 16\ 234$. Speedups are relative to the sequential execution time of 0.108 s of a radix-2 FFT program.

1. A set of $p-1$ splitter keys is selected. Conceptually, the splitters will partition the input data into p buckets.
2. All keys assigned to the ith bucket are sent to the ith processor.
3. Each processor sorts its bucket.

The selection of splitters that define approximately equal-sized buckets is a crucial issue. The standard approach is to randomly select pr keys from the input set, where r is called the oversampling ratio. These keys are sorted, and the keys with ranks $r, 2r, 3r, \ldots, (p-1)r$ are selected as the splitters. By choosing a large enough oversampling ratio, it can be shown with high probability that no bucket will contain many more keys than the average [27].

Since randomised sample sort is suitable for general-purpose parallel computing, it is not surprising that the approach has been used in numerous experimental studies. Blelloch et al. [8] describe randomised sample sort experiments on a Connection Machine CM-2. Hightower et al. [20] use randomised sample sort on a MasPar MP-1. Dusseau et al. [13] implement randomised sample sort on a Connection Machine CM-5 using the Split-C programming language [10] and the LogP cost model [11]. Randomised sample sort has also been implemented in direct BSP style by Juurlink and Wijshoff [28], Shumaker and Goudreau [37], and Hill et al. [23].

In terms of a *BSPlib* implementation, randomised sample sort is interesting in that the sending of keys to appropriate buckets requires irregular communication. For such routing patters, BSMP is a natural communication approach.

The piece of code below shows how the input data is sent to the appropriate buckets. The input data is stored in the array of doubles `a` of size `a_size`. The splitters have already been selected and distributed, and are stored in array `s` of size `p - 1`. The function bucket( ) takes a double and the array `s` and returns the appropriate bucket to send the double to. Each process will store its bucket in array `b` of size `b_size`.

```
/* Distribute each element of a [ ] to appropriate processor */
for(i=0; i<a_size; i++) {
  j=bucket(a[i], s, p-1);
  bsp_send(j, NULL, &a[i], sizeof(double));
}
bsp_sync( );
/* Queue size needed to allocate bucket (dummy not used) . */
bsp_qsize(&b_size, &dummy);
/* Allocate memory for bucket. */
b=(double*) calloc(b_size, sizeof(double));
/* Read in messages, store in b[ ]. */
for(i=0; i<b_size; i++)
  bsp_move(&b[i], sizeof(double));
```

Some experimental results on an SGI Power Challenge with 16 MIPS R4400 processors are shown in Fig. 2. The Power Challenge is a shared memory platform. The figure shows the speedup relative to the standard C library `qsort` (quicksort) run on

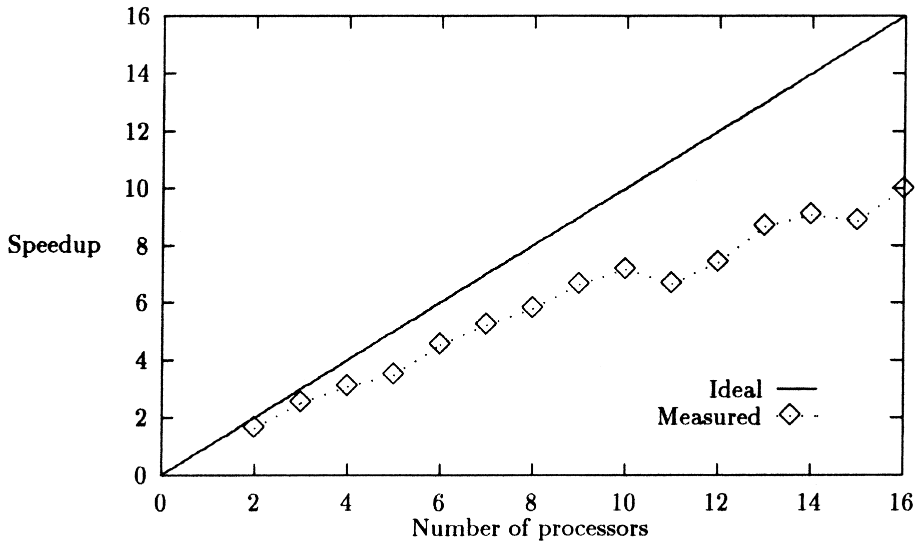

Fig. 2. Absolute speedup for a randomised sample sort on an SGI Challenge with 16 processors. The problem size is $n = 10^7$ double-precision floating point numbers with an oversampling ratio of $r = 100$. Speedups are relative to the sequential execution time of 128.04 s for the C standard library `qsort` function.

one processor. The input data was randomly generated. Only one test was run in each case.

5.4. Barnes–Hut N-body algorithm

The N-body problem is the problem of simulating the movement of a set of N bodies under the influence of gravitational, electrostatic, or other type of force. The problem has numerous applications, e.g. in astrophysics, molecular dynamics, fluid dynamics, and even computer graphics. The basic approach taken by most N-body algorithms is to simulate the system in discrete time steps, thus reducing the problem to that of computing the forces among the N bodies. While this can be done for long-range forces in a straightforward way by computing all N^2 pairwise interactions, several more efficient algorithms have been proposed that use a tree data structure to approximate the forces among N bodies in close to linear time, e.g., see Refs. [1,2,18,42].

In this section, we describe a *BSPlib* implementation of the Barnes–Hut algorithm [2], which achieves a running time of $O(N \log N)$ (under certain assumptions about the input distribution). The algorithm first inserts all bodies into an oct-tree structure, such that no leaf of the tree contains more than some fixed number of bodies. Then an upward pass through the tree is performed to compute the centre of mass (or some higher-order approximation of the mass distribution) of the bodies in each subtree. Finally, the force exerted on each body is approximated by performing a truncated depth-first traversal of the oct-tree, during which the force due to a sufficiently far away cluster of bodies is approximated using the centre of mass (or higher-order approximation) of the corresponding subtree.

In our parallel implementation, we use Orthogonal Recursive Bisection (ORB) to partition the domain into rectangular regions. Each processor first constructs an oct-tree locally by inserting all bodies that are located in its computational domain and computing the centres of mass of that tree. Then appropriate subtrees, called *locally essential trees*, are exchanged between the processors, using a replication scheme similar to those of Warren and Salmon [41] and Liu and Bhatt [29]. Afterwards, every processor has a local oct-tree that contains all the data needed to perform the tree-traversal on its bodies, and whose structure is consistent with that of the global oct-tree constructed by the sequential algorithm. The *BSPlib* implementation was obtained by porting a code originally written for the Green BSP library. A more detailed description of the replication scheme and its extension to other N-body algorithms can be found in Ref. [7].

The Barnes–Hut implementation is a good example for the use of the message passing primitives in *BSPlib*. As said before, BSMP often has advantages over DRMA for applications that manipulate irregular data structures, such as the oct-tree structure in the Barnes–Hut algorithm. During the replication phase of the parallel implementation, processors use a sender-driven protocol to send out all data that is needed by another processor. At the receiving end, the data is inserted back into the local tree structure. As the final format and destination of the data depends on the locally held data, it would be difficult to implement this replication phase with DRMA operations.

Our implementation performs only six supersteps per iteration; this makes the program efficient even on fairly small problem sizes and high-latency platforms. The application is irregular and dynamic, due to the changing positions of the bodies. However, the load distribution can be predicted fairly accurately from that of the previous iteration, as the system evolves only slowly. Under certain uniformity assumptions, the size of the h-relation in the replication phase is $O(p + n^{2/3})$ where n is the number of particles per processor. For reasonably large n, these bandwidth requirements are fairly modest, as we were careful in minimising the amount of data sent during the replication phase.

Fig. 3 shows the speedup of the Barnes–Hut code on a 16-processor SGI Challenge shared memory machine. The timings were obtained by running several iterations of the Barnes–Hut algorithm, and taking the average running time over all except the first two iterations. We used the separation parameter $\theta = 0.5$ and centre of mass approximations, and allowed up to 40 bodies in a leaf of the oct-tree. The reported speedups are relative to the running time of our code on a single processor (which incurs none of the parallel overheads, and which we believe to be a reasonable sequential implementation of the Barnes–Hut algorithm).

For our smallest input size (4000 particles), we observe that the speedup increases to a peak of about 9.5 on 13 processors, after which it stays essentially flat up to 16

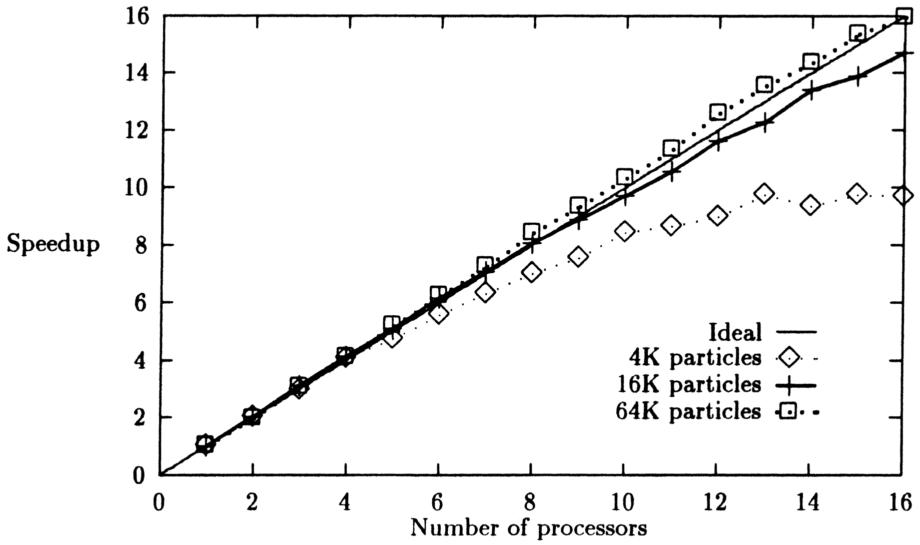

Fig. 3. Relative speedup for a molecular dynamics computation on an SGI Challenge with 16 processors. Speedups are relative to the parallel program with $p = 1$, which takes 13.64 s per iteration for 4000 particles, 102.32 s for 16 000, and 474.7 s for 64 000.

processors. [3] This behaviour can be explained by the fact that for small input sizes, the cost of the replication phase is relatively large. We point out that 4000 particles is indeed a very small input size in many simulations, and that the replication phase was not designed with this case in mind.

Fig. 3 also shows that the speedup quickly improves as we increase the input size. In particular, for 64 000 particles, we obtain essentially linear speedup (15.93 on 16 processors). The speedup is actually slightly superlinear for many data points, probably due to cache effects.

6. Conclusions and future work

This work has identified a complete set of 20 core primitives, or *level 0 operations*, that are needed to write parallel programs conveniently in BSP style. Together, these operations define *BSPlib*; an overview is given in Table 2. The limited size of *BSPlib* makes it easy to learn how to use the library, and also relatively easy to implement the library on a new architecture. In particular, this will help hardware and system

[3] The slight variations between 13 and 16 processors are probably due to the different partitionings of the data under our partitioning scheme.

Table 2
A quick reference to the 20 primitives of BSPlib

Class	Primitive	Meaning	Return type	Parameters	See sections
SPMD	`bsp_begin`	Start of SPMD part		`int maxprocs`	2.1
	`bsp_end`	End of SPMD part			2.1
	`bsp_init`	Simulate dynamic processes		`void(*spmd_part)(void), int argc, char *argv[]`	2.2
	`bsp_abort`	One process halts all		`char *format,...`	2.3
	`bsp_nprocs`	Number of processes	`int`		2.4
	`bsp_pid`	My process identifier	`int`		2.4
	`bsp_time`	Elapsed local time	`double`		2.4
	`bsp_sync`	Barrier synchronisation			2.5
DRMA	`bsp_pushreg`	Make area globally visible		`const void *ident, int size`	3.1
	`bsp_pop_reg`	Remove global visibility		`const void *ident`	3.1
	`bsp_put`	Copy to remote memory		`int pid, const void *src, void *dst, int offset, int nbytes`	3.2
	`bsp_hpput`	Unbuffered put		`int pid, const void *src, void *dst, int offset, int nbytes`	3.2
	`bsp_get`	Copy from remote memory		`int pid, const void *src, int offset, void *dst, int nbytes`	3.3
	`bsp_hpget`	Unbuffered get		`int pid, const void *src, int offset, void *dst, int nbytes`	3.3
BSMP	`bsp_set_tag-size`	Set tag size		`int *tag_nbytes`	4.1
	`bsp_send`	Send to remote queue		`int pid, const void *tag, const void *payld, int payld_nbytes`	4.2
	`bsp_qsize`	Number of messages in queue		`int *nmessages, int *accum_nbytes`	4.3
	`bsp_get_tag`	Get message tag		`int *status, void *tag`	4.4
	`bsp_move`	Move message from queue		`void *payload, int reception_nbytes`	4.5
	`bsp_hpmove`	High performance move	`int`	`void **tag_ptr, void **payload_ptr`	4.6

An empty field for return type or parameters denotes `void`.

software developers in focusing their efforts to design efficient implementations of a communications library for their systems.

Some message passing systems, such as MPI [34,19], provide operations for various specialised collective communication patterns which arise frequently in message passing programs. These include broadcast, scatter, gather, total exchange, reduction (fold), prefix sums (scan), etc. These standard communication patterns also arise frequently in the design of BSP algorithms. It is important that such structured patterns can be conveniently expressed and efficiently implemented in a BSP programming system, in addition to the more primitive operations such as put, get, and send which generate arbitrary and unstructured communication patterns.

The library we have described can easily be extended to support such structured communications by adding `bsp_bcast`, `bsp_fold`, `bsp_scan`, `bsp_gather`, `bsp_scatter`, `bsp_exchange`, etc. as higher level operations. We call these *level 1 operations*. Such operations can be implemented in terms of the level 0 operations, or directly on top of the architecture if that is more efficient. For modularity and safety, all level 1 operations have the following semantics: (1) Remote memory areas that are accessed by DRMA operations must be registered and deregistered within the level 1 operation. Registration must be followed by a synchronisation; for de-registration this is not necessary. (2) The tag size of messages sent by a BSMP operation must be set within the level 1 operation. This requires a synchronisation. The tag size must be reset to the previous value on exit. This does not require a synchronisation. (3) The messages issued by a BSMP operation must be delivered within the level 1 operation. This requires a synchronisation. They must be moved from the system queue, which must be empty on exit. This does not require a synchronisation.

We have not included level 1 operations in the *BSPlib* definition, since this would lead to a proliferation of primitives, which in turn would diminish the focus provided by a small size of the library. Furthermore, it is still unclear which level 1 operations are really useful in applications; in different application areas there may be a need for many different types of operations.

BSPlib finalises the definition process of a BSP communications library, and it is unlikely to be changed in the future. The work that remains to be done in the future includes, first of all, developing more and better implementations of *BSPlib*. In particular, we pose the challenge to hardware vendors to provide good implementations that are characterised by high values of s but low values of g and l. Furthermore, there is much work to be done in the area of level 1 operations. We envision a situation where application developers will release separate level 1 libraries containing those operations they found useful. For reasons of portability, such libraries should be formulated in terms of the level 0 operations. Once a consensus emerges about the important level 1 operations, some hardware or software developers could take the opportunity to provide added value or distinguish their products by developing more efficient implementations directly on top of the hardware. Finally, other directions in which *BSPlib* could be extended are parallel I/O and fault tolerance. This could again be done by developing separate libraries.

Acknowledgements

The works of Jonathan Hill and Bill McColl were supported in part by the EPSRC Portable Software Tools for Parallel Architectures Initiative, as Research Grant GR/K40765 "A BSP Programming Environment", October 1995–September 1998. Rob Bisseling would like to thank The Netherlands Computer Facilities foundation and the High Performance Applied Computing centre at Delft University of Technology for provided access to a Cray T3E.

The authors would like to thank Paul Crumpton, Alex Gerbessiotis, Tony Hoare, Antoine Le Hyaric, Tim Lanfear, Bob McLatchie, Richard Miller, David Skillicorn, Bolek Szymanski, Leslie Valiant, and Hong Xie for various discussions on BSP libraries. Thanks also to Jim Davies for improving the typesetting in $\TeX$.

References

[1] A.W. Appel, An efficient program for many-body simulation, SIAM Journal on Scientific and Statistical Computing 6 (1) (1985) 85–103.

[2] J. Barnes, P. Hut, A hierarchical $O(N \log N)$ force-calculation algorithm, Nature 324 (1986) 446–449.

[3] R. Barriuso, A. Knies, SHMEM User's Guide, Revision 2.0, Cray Research Inc., Mendota Heights, MN, May 1994.

[4] R.H. Bisseling, Sparse matrix computations on bulk synchronous parallel computers, in: G. Alefeld, O. Mahrenholtz, R. Mennicken (Eds.), ICIAM '95. Issue 1. Numerical Analysis, Scientific Computing, Computer Science, Akademie, Berlin, 1996, pp. 127–130.

[5] R.H. Bisseling, Basic techniques for numerical linear algebra on bulk synchronous parallel computers, in: L. Vulkov, J. Waśniewski, P. Yalamov (Eds.), Workshop Numerical Analysis and its Applications 1996, Lecture Notes in Computer Science, vol. 1196, Springer, Berlin, 1997, pp. 46–57.

[6] R.H. Bisseling, W.F. McColl, Scientific computing on bulk synchronous parallel architectures, in: B. Pehrson, I. Simon (Eds.), Technology and Foundations: Information Processing'94, vol. I, IFIP Transactions A, vol. 51, Elsevier, Amsterdam, 1994, pp. 509–514.

[7] D. Blackston, T. Suel, Highly portable and efficient implementations of parallel adaptive N-body methods, in: Supercomputing'97, November 1997.

[8] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, M. Zagha, A comparison of sorting algorithms on the connection machine CM-2, in: Third Annual ACM Symposium on Parallel Algorithms and Architectures, 1991, pp. 3–16.

[9] T. Cheatham, A. Fahmy, D.C. Stefanescu, L.G. Valiant, Bulk synchronous parallel computing – a paradigm for transportable software, in: 28th Hawaii International Conference on System Science, vol. II, IEEE Computer Society Press, Silver Spring, MD, January 1995, pp. 268–275.

[10] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, Parallel programming in Split-C, in: Supercomputing '93, November 1993.

[11] D. Culler, R. Karp, D.Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: towards a realistic model of parallel computation. in: Fourth ACM Symposium on Principles and Practice of Parallel Programming, May 1993, pp. 1–12.

[12] S.R. Donaldson, J.M.D. Hill, D.B. Skillicorn, Predictable communication on unpredictable networks: implementing BSP over TCP/IP, in: EuroPar'98, Southampton, UK, September 1998, Lecture Notes in Computer Science, Springer, Berlin.

[13] A.C. Dusseau, D.E. Culler, K.E. Schauser, R.P. Martin, Fast parallel sorting under LogP: experience with the CM-5, IEEE Transactions on Parallel and Distributed Systems 7 (8) (1996) 791–805.

[14] W.D. Frazer, A.C. McKellar, Samplesort: a sampling approach to minimal storage tree sorting, Journal of the ACM 17 (3) (1970) 496–507.
[15] A.V. Gerbessiotis, L.G. Valiant, Direct bulk-synchronous parallel algorithms, Journal of Parallel and Distributed Computing 22 (2) (1994) 251–267.
[16] M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, Towards efficiency and portability: programming with the BSP model, in: Eighth Annual ACM Symposium on Parallel Algorithms and Architectures, June 1996, pp. 1–12.
[17] M.W. Goudreau, K. Lang, S.B. Rao, T. Tsantilas, The Green BSP Library, Technical Report CS-TR-95-11, Department of Computer Science, University of Central Florida, Orlando, FL, June 1995.
[18] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, Journal of Computational Physics 73 (2) (1987) 325–348.
[19] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, MA, 1994.
[20] W.L. Hightower, J.F. Prins, J.H. Reif, Implementations of randomized sorting on large parallel machines, in: Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pp. 158–167.
[21] J.M.D. Hill, P.I. Crumpton, D.A. Burgess, Theory, practice, and a tool for BSP performance prediction, in: EuroPar '96, Lecture Notes in Computer Science, vol. 1124, Springer, Berlin, August 1996, pp. 697–705.
[22] J.M.D. Hill, S. Donaldson, D. Skillicorn, Stability of communication performance in practice: from the Cray T3E to Networks of Workstations, Technical Report PRG-TR-33-97, Programming Research Group, Oxford University Computing Laboratory, October 1997.
[23] J.M.D. Hill, S.R. Donaldson, D.B. Skillicorn, Portability of performance with the BSPlib communications library, in: Programming Models for Massively Parallel Computers (MPPM'97), London, November 1997, IEEE Computer Society Press, Silver Spring, MD.
[24] J.M.D. Hill, B. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, R. Bisseling, BSPlib: The BSP Programming Library, Technical Report PRG-TR-29-9, Oxford University Computing Laboratory, Oxford, UK, May 1997. See `www.bsp-worldwide.org` for more details.
[25] J.M.D. Hill, D. Skillicorn, Lessons learned from implementing BSP, Future Generation Computer Systems 13 (4/5) (1998) 327–335.
[26] C.A.R. Hoare, Quicksort, Computer Journal 5 (1) (1962) 10–15.
[27] J.S. Huang, Y.C. Chow, Parallel sorting and data partitioning by sampling, in: Seventh International Computer Software and Applications Conference (COMPSAC'83), IEEE Computer Society, November 1983, pp. 627–631.
[28] B.H.H. Juurlink, H.A.G. Wijshoff, A quantitative comparison of parallel computation models, in: Eighth Annual ACM Symposium on Parallel Algorithms, and Architectures,June 1996, pp. 13–24.
[29] P. Liu, S.N. Bhatt, Experiences with parallel N-body simulations, in: Sixth Annual ACM Symposium on Parallel Algorithms and Architectures, 1994, pp. 122–131.
[30] W.F. McColl, Scalable computing, in: J. van Leeuwen (Ed.), Computer Science Today: Recent Trends and Developments, Lecture Notes in Computer Science, vol. 1000, Springer, Berlin, 1995, pp. 46–61.
[31] W.F. McColl, Scalability, portability and predictability: the BSP approach to parallel programming, Future Generation Computer Systems 12 (4) (1996) 265–272.
[32] R. Miller, A library for bulk synchronous parallel programming, in: Proceedings of The BCS Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing, December 1993, pp. 100–108.
[33] P.B. Monk, A.K. Parrott, P.J. Wesson, A parallel finite element method for electro-magnetic scattering, COMPEL 13 (Supp. A) (1994) 237–242.
[34] MPI: A Message Passing Interface, version 2.0, Message Passing Interface Forum, July 1997.
[35] C.H. Papadimitriou, M. Yannakakis, Towards an architecture-independent analysis of parallel algorithms, in: 20th ACM Symposium on Theory of Computing, ACM, 1998, pp. 510–513.
[36] J.H. Reif, L.G. Valiant, A logarithmic time sort for linear size networks, Journal of the ACM 34 (1) (1987) 60–76.

[37] G. Shumaker, M.W. Goudreau, Bulk-synchronous parallel computing on the Maspar, in: World Multiconference on Systemics, Cybernetics and Informatics, vol. 1, July 1997, pp. 475–481.
[38] D. Skillicorn, J.M.D. Hill, W.F. McColl, Questions and answers about BSP, Scientific Programming 6 (3) (1997) 249–274.
[39] L.G. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990) 103–111.
[40] C. Van Loan, Computational Frameworks for the Fast Fourier Transform, Frontiers in Applied Mathematics, vol. 10, SIAM, Philadelphia, PA, 1992.
[41] M.S. Warren, J.K. Salmon, Astrophysical N-body simulations using hierarchical tree data structures, in: Supercomputing'92, 1992, pp. 570–576.
[42] F. Zhao, An O(N) algorithm for three-dimensional N-body simulations, Technical Report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, October 1987.