A Proposal for the BSP Worldwide Standard Library (preliminary version)

Mark W. Goudreau[1]
Jonathan M. D. Hill[2]
Kevin Lang[3]
Bill McColl[2]
Satish B. Rao[3,4]
Dan C. Stefanescu[5]
Torsten Suel[3,4]
Thanasis Tsantilas[6]

Authors' affiliations:
(1) University of Central Florida
(2) Oxford University
(3) NEC Research Institute, Princeton
(4) University of California at Berkeley
(5) Harvard University
(6) Columbia University

Information concerning BSP Worldwide can be found at:
`http://www.bsp-worldwide.org/`

Date of publication: April 1996

Contents

Things to do

1. discuss what can be expected from I/O
2. collective communications
3. discuss DRMA operations in MPI2
4. add useful indexing

Chapter 1

Introduction

A bulk synchronous parallel (BSP) computer [7, 11] consists of a set of processor-memory pairs, a global communications network, and a mechanism for the efficient barrier synchronisation of the processors. There are no specialised broadcasting or combining facilities, although these can be efficiently realised in software where required. The model also does not deal directly with issues such as input-output or the use of vector units, although it can be easily extended to do so.

A BSP computer operates in the following way. A computation consists of a sequence of parallel supersteps. During a superstep, each processor-memory pair can perform a number of computation steps on values held locally at the start of the superstep, send and receive a number of messages, and handle various remote get and put requests. The model does not prescribe any particular style of communication, although at the end of a superstep there is a barrier synchronisation at which point any pending communications must be completed.

Although we have described the BSP computer as an architectural model, one can also view bulk synchrony as a programming discipline. The essence of the BSP approach to parallel programming is the notion of the superstep, in which communication and synchronisation are completely decoupled. A BSP program is simply one which proceeds in phases, with the necessary global communications taking place between the phases. 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 software for scalable computing.

1.1 BSP programming languages and libraries

The communication facilities described in this proposal are drawn from other communication libraries that provide either direct remote memory access or message passing facilities. In this section we briefly summarise the work that has influenced this proposal.

The PVM message passing library [3] is widely implemented and widely used. The MPI message passing interface [9] is more elaborate. It supports blocking and non-blocking point-to-point communication and a number of collective communications (broadcast, scatter, gather, reduction, etc.). Although neither of these libraries is directly aimed at supporting BSP programming, they can be used for that purpose.

The Cray SHMEM library provides primitives for direct remote memory access (or one-

sided communications). The Oxford BSP Library [10] provides a similar set of programming primitives for bulk synchronous remote memory access. The core of the library consists of six routines—two for process management, two for superstep synchronisation, and two for remote memory access. Higher level operations such as broadcast and reduction are also available.

The Green BSP Library [5, 4] provides a set of message passing primitives for exchanging fixed sized packets between processes. In contrast to synchronous message passing, the sending and reception of messages is decoupled. In accordance with superstep semantics, the messages are guaranteed to be delivered by the end of a superstep—we term this bulk synchronous message passing.

Split-C [2] is a parallel extension of C which supports efficient access to a global address space on current distributed memory architectures. It aims to support careful engineering and optimisation of portable parallel programs by providing a cost model for performance prediction. The language extension is based on a small set of global access primitives and simple memory management declarations which support both bulk synchronous and message driven styles of parallel programming. There are several other BSP programming languages under development as part of research projects, such as GPL [8] and Opal [6] at Oxford and BSP-L [1] at Harvard. We will not discuss those here.

1.2 The BSP communication spectrum

A wide variety of communication operations can be used in a BSP computation. There is a *spectrum* between the leanest and most efficient mechanisms, and those which are more convenient for the programmer. At one end of the spectrum, unbuffered bulk synchronous direct remote memory access (DRMA) provides the highest performance. However it requires the programmer to consider the existence and location of data structures on the destination process where data is communicated into, and to ensure that the data on the process initiating the DRMA is not changed during the lifetime of the superstep (see §2.3.2). At the other end of the spectrum, buffered bulk synchronous message passing (BSMP) is more flexible, as it relieves the programmer from both of these requirements. There is no need to consider the existence or placement of data at the destination of the communication, as BSMP communicates into a system buffer from where messages are copied from by the user at the end of the superstep. There is also no need to consider if the data on the initiating process is changed during the lifetime of the superstep as it is buffered.

How large are the efficiency differences across the BSP spectrum from unbuffered DRMA to buffered BSMP? Any overheads with BSMP will be due to the buffering of communication at the sending and destination processes. This can be quantified by considering that if g is the asymptotic communication bandwidth for a DRMA operation, then $g + 2m$ is the asymptotic communication bandwidth for a BSMP; where m is the cost of writing a single word into main memory. Since m will never be larger than g, the ratio between the costs of BSMP and DRMA will never be more than three. For most machines, the ratio will be much less than this.

In the next chapter we present a proposal for the BSP Worldwide Standard Library. The library supports both bulk synchronous remote memory access and bulk synchronous message passing. The set of basic operations is given in Table 1.1. A higher level library, described in Appendix A, provides various specialised collective communication operations. These are not considered as part of the core library, as they can be easily realised in terms of the core.

Class	Operation	See section
Initialisation	`bsp_init`	§2.1
	`bsp_begin`	§2.1
	`bsp_end`	§2.1
Enquiry	`bsp_pid`	§2.1
	`bsp_nprocs`	§2.1
	`bsp_time`	§2.6
Synchronisation	`bsp_sync`	§2.2
DRMA	`bsp_register`	§2.3.1
	`bsp_put`	§2.3.3
	`bsp_get`	§2.3.4
BSMP	`bsp_set_tag_size`	§2.4.1
	`bsp_send`	§2.4.2
	`bsp_get_tag`	§2.4.3
	`bsp_move`	§2.4.3
Halt	`bsp_abort`	§2.5
High Performance	`bsp_hpput`	§2.3.3
	`bsp_hpget`	§2.3.4
	`bsp_hpmove`	§2.4.3

Table 1.1: Core BSP operations

Chapter 2

Proposal

This proposal can be viewed as an attempt to achieve a synthesis of the various approaches to low level BSP programming which are currently being pursued. Our main concern when defining the semantics for each of the library operations was to provide as general a semantics as possible, whilst ensuring that the implementor has the greatest possible scope to provide an efficient implementation of the library on the various forms of parallel architecture. For simplicity, we will refer to the library as *BSPlib*.

2.1 Creating BSP processes

Processes are created in a *BSPlib* program by the operations `bsp_begin` and `bsp_end`[1] as defined in figure 2.1. 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` pair within a program, although there are two different ways to start a *BSPlib* 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. It can then spawn the required number of processes using `bsp_begin`. Execution of the spawned processes then continue 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 the second of these modes is that some parallel machines available today[2] do not provide dynamic process creation. As a solution to this problem we provide the second mode by making such machines *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

[1] as a naming convention, all *BSPlib* C procedures have an underscore within them, whereas in FORTRAN we use the same name without an underscore.

[2] almost all distributed memory machines, e.g. IBM SP2, Cray T3D, Meiko CS-2, Parsytec GC, Hitachi SR2001.

```
void bsp_init(void (*startproc)(void));

void bsp_begin(int maxprocs);

void bsp_end()
```

```
SUBROUTINE bspinit(startproc)
   INTERFACE
      SUBROUTINE startproc
   END INTERFACE

SUBROUTINE bspbegin(maxprocs)
   INTEGER, intent(IN)::maxprocs

SUBROUTINE bspend()
```

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

`startproc` is the name of a procedure that contains `bsp_begin` and `bsp_end` as its first and last statements.

Figure 2.1: Procedures to begin and end a BSP computation

`bsp_begin` and `bsp_end` as its first and last statements. The rationale for this approach is described in § 3.1.

Figure 2.2 defines a pair of functions that determine the total number of processes and the number used to identify an individual process. 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 n, the actual number of processes allocated to the program, where $1 \leq n \leq maxprocs$, and $maxprocs$ is the number of processes requested in `bsp_begin`. Each of the n processes created by `bsp_begin` has a unique value m in the range $0 \leq m \leq n-1$. The function `bsp_pid` returns the value of the process executing the function call.

```
int  bsp_nprocs();

int  bsp_pid();
```

```
INTEGER FUNCTION bspnprocs()

INTEGER FUNCTION bsppid()
```

Figure 2.2: Enquiry functions

Notes

1. 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.

2. 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.

3. 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.

4. `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.

5. If the program is not run in a purely SPMD mode, then `bsp_init` has to be the first statement executed by the program.

6. `bsp_begin(bsp_nprocs())` can be used to request the number of processes as there are processors on a parallel machine.

2.2 Superstep synchronisation

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 superstep and may communicate data to other processes. Any communications within a superstep are guaranteed to occur by the end of the superstep, where the `bsp_nprocs` processes synchronise at a barrier—*BSPlib* has no form of subset synchronisation (see §3.2). The end of one superstep and the start of the next is identified by a call to the library procedure `bsp_sync` as defined in Figure 2.3.

The semantics of `bsp_sync` could be modified to allow a process that has reached `bsp_end` to implicitly meet all superstep barriers with its siblings. We have decided not to include this in the current proposal. If it were included, one might want to add a stronger synchronisation primitive which had the same requirement, i.e., full participation, as we have defined for `bsp_sync` here.

```
void bsp_sync();                    |   SUBROUTINE bspsync()
```

where any communication occurring between two successive calls to `bsp_sync` takes effect after the latter `bsp_sync`.

Figure 2.3: Procedure for barrier synchronisation

2.3 Bulk synchronous remote memory access

One way of performing data communication in the BSP model is to use Direct Remote Memory Access (DRMA) communication facilities. The DRMA operations available in the Cray SHMEM and Oxford BSP libraries require that the communicated data structures are held in statically allocated memory locations. In *BSPlib* the DRMA operations are also well defined for stack and heap allocated data structures and for heterogeneous environments. This is achieved by only 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_register` to announce the address of the start of their 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 address space of another process, without the involvement of that process, is potentially dangerous. The mechanisms we propose here exhibit different degrees of *safety*. The right choice depends upon the class of applications and the desired goals, and has to be made by the user (see §2.3.2 for more details).

2.3.1 Making local areas available for remote use

```
void bsp_register(const void *ident);
```

```
SUBROUTINE bspregister(ident)
   <TYPE>, intent(IN) :: ident
```

where `ident` is a previously initialized variable denoting the name of the local area being registered

Figure 2.4: Procedure for 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. 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 physical addresses.

To allow *BSPlib* programs to execute correctly in such a setting we require a mechanism for relating these various addresses. During a superstep, each process may produce an ordered sequence of registration operations. The length of this sequence must be the same for all processes (if not, then an error is raised). For a superstep where the length is k, let $ident_i^j$ denote the argument of the j^{th} registration operation in process i; where $0 \leq i < p$ and $0 \leq j < k$. The registration mechanism allows any DRMA operation to address the remote area $ident_r^j$ on process r by giving the corresponding area $ident_l^j$ on the local process l. More formally, given a call to `bsp_register`($ident_l^j$), every process i (where $0 \leq i < p$) associates with its local reference $ident_l^j$ a mapping of the form $ident_l^j \mapsto \{ident_0^j, \ldots, ident_{p-1}^j\}$. When a DRMA put operation (see §2.3.3) is performed on process l with the following arguments:

$$\texttt{bsp\_put}(r, \texttt{src}, \texttt{tgt}_l^j, \texttt{offset}, \texttt{nbytes});$$

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:

$$\texttt{tgt}_r^j + \texttt{offset}$$

on process `r`; where the base address $\texttt{tgt}_l^j$ is a registered memory area of the form $\texttt{tgt}_l^j \mapsto \{\texttt{tgt}_0^j, \ldots, \texttt{tgt}_{p-1}^j\}$.

Notes

1. Registration takes effect at the end of the superstep. DRMA operations may use the registered areas from the start of the next superstep.

2. Registration information can be changed at any subsequent point in the program by a further call to `bsp_register`. If no such re-registration is performed then the registration persists until the end of the program.

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_register` although some may register the memory area `NULL`[3]. This memory area is reguarded as unregistered.

6. The explicit registration mechanism removes possible implicit assumptions about the compilation of `static` data as used by the Cray SHMEM and Oxford BSP libraries. It should be noted that static data structures will always have the simple registration information:

$$ident_l^j \mapsto \{\underbrace{ident_l^j, \ldots, ident_l^j}_{p \text{ copies}}\}$$

since, on each processor, static data structures are allocated at the same address[4].

2.3.2 Buffering semantics of DRMA operations

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

Buffered remotely: Data communication into registered areas will only take effect *at* the end of the superstep. Therefore, the registered remote area may be safely operated on during a superstep, although any changes will be overwritten at the end of a superstep if data is communicated there.

Unbuffered remotely: Data communication into registered areas can take effect at any time during the superstep.

Buffered locally: The data communicated from a process will contain the values which are held at the time the operation was called[5]. Therefore, the process may reuse its storage area during the superstep.

Unbuffered locally: The data transfer resulting from a call to a communication operation may occur at any time between the time of the call and the end of the superstep. Therefore, for safety, no process should change the data structures used in this communication 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 computation taking place on the processors involved in the operation. However, it should be noted that even the most cautious choice of buffering mode does not completely remove the effects of non-determinism from a remote process. For example, if more than one process transfers data into overlapping memory locations, then the result at the overlapping region will be nondeterministically chosen; it is implementation dependent which one of the many "colliding" communications should be written into the remote memory area.

[3] the array `bspunregistered` may be used by FORTRAN programmers

[4] this isn't always the case, as some optimising C compilers *un-static* statics.

[5] more precisely, this is the time when the communication operation is apparent to the process holding the data.

```
void bsp_[hp]put(
              int pid,
              const void *src,
              void *dst,
              int   offset,
              int   nbytes);
```

```
SUBROUTINE bsp[hp]put(pid,src,dst,offset,nbytes)
   INTEGER, intent(IN) :: pid, offset, nbytes
   <TYPE>,  intent(IN) :: src, dst
```

where `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 data 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 size which is a multiple of the word size of the machine.

Figure 2.5: Procedure for *put* and *hpput*

2.3.3 Putting data into a remote location

The aim of `bsp_put` and `bsp_hpput` is to provide an operation akin to `memcpy(3C)` 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 adopted for *BSPlib* `bsp_put` communication (see Figure 2.5) is *buffered locally/buffered remotely*. While the semantics is clean and safety is maximized, puts may tax unduly the memory resources of an implementation, thus preventing large transports of data (see §3.3 for more details). Consequently, *BSPlib* also provides a *high performance put* operation `bsp_hpput` whose semantics is *unbuffered locally/unbuffered remotely*. The use of this operation requires care as correct data delivery is only guaranteed if neither communication nor local/remote computations modify either the source or the destination areas. 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.

2.3.4 Getting data from a remote location

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

```
void bsp_[hp]get(
            int   pid,
            const void *src,
            int   offset,
            void *dst,
            int   nbytes);
```

```
SUBROUTINE bsp[hp]get(pid,src,offset,dst,nbytes)
   INTEGER, intent(IN)  :: pid, nbytes, offset
   <TYPE>   intent(IN)  :: src, dst
```

where `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 from. `src` must be a previously registered data-structure.

`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 data structures that are at least `nbytes` in size.

Figure 2.6: Procedure for *get* and *hpget*

of the process that initiated them, see Figure 2.6 for the definition of `bsp_get` and `bsp_hpget`.

The semantics adopted for *BSPlib* `bsp_get` communication is *buffered locally/buffered remotely*. This semantics means that the value taken from the source on the remote process by the get, is the value *at* the end of the superstep. Consequently the value written from the remote process into the destination memory area on the initiating process only takes effect at the end of the superstep as well.

A high-performance version of get, `bsp_hpget`, provides an *unbuffered locally/unbuffered remotely* semantics in which the two-way communication can take effect at anytime during the superstep.

2.4 Bulk synchronous message passing

Bulk synchronous remote memory access 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 between 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 bulk synchronous message passing (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, a collection of messages sent to the same process has no implied ordering at the receiving end. However, since each message may be tagged, the programmer can identify messages by their tag.

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*. Both parts can be of arbitrary type, but we expect that in FORTRAN programs the tag will almost always be an array of integers while the payload will often be an array of reals. In C programs, either part could also be a complicated structure. 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 locally/buffered remotely*. We note that this buffering classification is a semantic description; it does not necessarily describe the underlying implementation.

2.4.1 Setting the 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. Moreover, in any superstep where `bsp_set_tag_size` is called, it must be called before sending any messages.

```
void bsp_set_tag_size (int *tag_bytes);
```

```
SUBROUTINE bspsettagsize(tag_bytes)
    INTEGER, intent(INOUT) :: tag_bytes
```

where `tag_bytes` , on entry to the procedure, specifies the size of the fixed-length portion of every message in the current and succeeding supersteps; the default tag size is zero. On return from the procedure, `tag_bytes` is changed to reflect the *previous* value of the tag size (see §3.5 for rationale).

Figure 2.7: Function for setting tag size

Notes

1. The tag size of incoming messages is prescribed by the outgoing tag size of the previous step.

2. `bsp_set_tag_size` must be called by *all* processes with the same argument in the same superstep. In this respect, it is similar in nature to a `bsp_register`.

3. `bsp_set_tag_size` must be called before any `bsp_send` in each process.

4. The default tag size is 0.

2.4.2 Sending a message

The `bsp_send` operation defined in Figure 2.8 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. It copies both the tag and the payload of the message out of user space into the system before returning. The `tag` and `payload` inputs are allowed to 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`.

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

```
SUBROUTINE bspsend(pid,tag,
                   payload,payload_bytes)
   INTEGER, intent(IN) :: pid
   <TYPE>,  intent(IN) :: tag
   <TYPE>,  intent(IN) :: payload
   INTEGER, intent(IN) :: payload_bytes
```

where `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_size_tag`.

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

`payload_bytes` is the size of the payload.

Figure 2.8: Procedure for sending messages

2.4.3 Procedures for receiving a message

To receive a message, the user should use the procedures `bsp_get_tag` and `bsp_move`. The operation `bsp_get_tag` defined in Figure 2.9 returns the tag of the first message in the buffer. The operation `bsp_move` defined in Figure 2.10 copies the payload of the first message in the buffer into `payload`, and removes that message from the buffer.

Note that `bsp_move` serves to flush the corresponding message from the buffer, 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.

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 the reception area size `reception_nbytes`, the payload will be truncated.

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

```
SUBROUTINE bspgettag(status, tag)
   INTEGER, intent(OUT) :: status
   <TYPE>,  intent(OUT) :: tag
```

where `status` becomes -1 if the system buffer is empty. Otherwise it becomes the length of the payload of the first message in the buffer. 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 buffer is empty. Otherwise it is assigned the tag of the first message in the buffer.

Figure 2.9: Procedure for getting tag of a message

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

```
SUBROUTINE bspmove(payload,reception_nbytes)
   <TYPE>,  intent(OUT) :: payload
   INTEGER, intent(IN)  :: reception_nbytes
```

where `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`.

Figure 2.10: Procedure for copying and then removing a message from the buffer

4. If `reception_nbytes` is zero this simply "removes" the message from the system buffer. This should be efficient in any implementation of the library.

2.4.4 A lean method for receiving a message

```
int bsp_hpmove(void ** tag_ptr_buf, void ** payload_ptr_buf);
```

where `bsp_hpmove` is a function which returns -1, if the system buffer is empty. Otherwise it returns the length of the payload of the first message in the buffer and (a) places a pointer to the tag in `tag_ptr_buf`; (b) places a pointer to the payload in `payload_ptr_buf`; and (c) removes the message (by advancing a pointer representing the head of the buffer).

Figure 2.11: High performance procedure for getting a message

The operation `bsp_hpmove` shown in Figure 2.11 is a non-copying method of receiving messages that is available in languages with pointers such as C, but not vanilla FORTRAN.

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. The storage referenced by these pointers remains valid until the end of the current superstep.

2.5 Raising an error and halting

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

```
SUBROUTINE bspabort(err_string)
   CHARACTER(*), intent(IN)::err_string
```

where `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.

`err_string` is single error string that is printed when the FORTRAN routine is executed. All computation ceases after a call to `bsp_abort`.

Figure 2.12: Procedure for halting a BSP computation

The function `bsp_abort` shown in Figure 2.12 can be used to 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. A single process in a potentially unique thread of control can therefore halt the entire *BSPlib* program.

2.6 Timing routine

```
double bsp_time();
```

```
DOUBLE PRECISION FUNCTION bsptime()
```

Figure 2.13: High-precision timing function

The function `bsp_time` defined in Figure 2.13 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 time in seconds since `bsp_begin`. The semantics of `bsp_time` is as though there were `bsp_nprocs` timers, one per process. *BSPlib* does *not impose any synchronisation requirements between the timers in each process.*

Chapter 3

Rationale

3.1 Simulating dynamic spawning of processes

The aim of the *BSPlib* initialisation routines is to provide the programmer with the freedom to calculate the number of processes required in a *BSPlib* computation before the SPMD processes are spawned. Unfortunately, this dynamic spawning mechanism does not fit well with many current distributed memory machines. On those machines, the number of processes involved in a computation has to be determined at the time that the *BSPlib* program is loaded onto the parallel machine. The purpose of the `bsp_init` primitive is to provide a simple method of simulating a dynamic spawning mechanism on all parallel architectures.

Consider two different kinds of parallel architecture: one that allows the dynamic creation of processes, and another that can only start a fixed number of processes at the beginning of a computation.

Dynamic creation In the example program shown in Figure 3.1, if a parallel machine supports dynamic process creation, then `bsp_init` has no effect on the program. A single thread of control is initiated at the start of the program, and requests an integer from standard-input that specifies the number of BSP processes to spawn. Up-to this number of processes are then spawned and a message is displayed on the terminal from each process. When `bsp_end` is executed, all but process zero is terminated, and the string `"I am process 0"` will be printed by the process that continues to execute the rest of the program sequentially.

Static creation If n processes are available when a *BSPlib* computation is loaded onto the parallel machine, then the effect of `bsp_init` is to make all but process zero call the zero-arity procedure specified by an argument to `bsp_init`. As `bsp_begin` has to be the first statement of that procedure, $n-1$ processes will idle in `bsp_begin` until the other process reaches that point in the code as well.

Process zero will return from `bsp_init` and execute the program in isolation of the other processes until it reaches `bsp_begin`. At that point it will know how many processes the user has requested. If that number is greater than n, then only n processes will be allotted to the user. If the number is less than n, then the processes not required by the *BSPlib* program will exit[1]. The program will then continue in a SPMD manner until `bsp_end` is encountered.

[1] if the parallel machine does not allow a subset of the processes to exit early, then they will idle to the end of the program.

```
int nprocs;

void spmd_start() {
  begin_bsp(nprocs);
    printf("Started process %d out of %d",bsp_pid(),bsp_nprocs());
  end_bsp();
}

void main() {
  bsp_init(spmd_start);
  nprocs = ReadInteger();
  printf("Going to try and start %d processes\n",nprocs);

  spmd_start();
  printf("I am process %d",bsp_pid());
}
```

Figure 3.1: An example *BSPlib* program

At that point all but one of the processes will be terminated[2], and process zero will continue to execute the rest of the program sequentially.

3.2 Why there is no subset synchronisation

Although it may sometimes appear to be convenient to have the possibility of groups of processes synchronising independently, we believe that this can significantly increase the difficulty of BSP cost analysis. The essence of BSP cost modeling is that the cost of a series of supersteps is simply the sum of the costs of each separate superstep. If subset synchronisation is allowed, then the cost of each group should probably be calculated independently (using its own p, g and l ?). The total cost of the computation will then be determined as some function of the costs of the groups (and their various p, g and l parameters?).

Over the last few years, various research projects on BSP languages have explored more flexible forms of synchronisation in BSP computations. The experience thus far suggests that in most cases, the advantages are modest or nonexistent, while the disadvantages (particularly in terms of cost analysis) are usually very considerable. We therefore feel that inclusion of subset synchronisation in this basic library would be unwise.

The wide variety of possible forms of subset synchronisation seems to provide another compelling reason for avoiding it at the core level. Should subset synchronisation be: static or dynamic?, if dynamic then on a superstep by superstep basis?, if static then one level of groups?, two levels?, many levels?, recursion and nested parallelism?, synchronisation across groups?, etc.

[2] again if a subset of the processes are not allowed to terminate early, then they will idle to the end of the program

3.3 Buffering of DRMA operations

Buffered DRMA operations maximise safety, ease the debugging process and simplify reasoning about programs. Indeed, a DRMA communication operation which is unbuffered remotely is potentially dangerous due to the added nondeterminism as data delivery can take effect at any time during a superstep. This puts a major burden on the programmer to determine when DRMA operations do not interfere with data that is being manipulated locally within a superstep. Furthermore, if a communication operation is unbuffered locally, the local data may be changed by remote communications or by local data manipulations, before being sent.

Thus, if BSP programs do not adhere to separating remote accesses from local manipulation of data, the added nondeterminism will complicate the programming and the debugging processes.

Buffering does not, however, come without cost. Each buffered operation requires twice the amount of memory of it's unbuffered alternative, and thus it limits the size of data that can be used in a given application. In turn, this may limit the usefulness of the library.

The solution adopted by *BSPlib* is to provide a set of maximally safe primitives, `bsp_put` and `bsp_get`, which can be used to develop and test *BSPlib* programs. These operations have a clean, simple semantics and allow a convenient entry point for new BSP programmers.

For high performance programming, *BSPlib* offers a set of fully unbuffered primitives, `bsp_hpput` and `bsp_hpget`, which have the potential of performing communication very efficiently, both in terms of time and space. Users can employ these operations from the outset or as replacements, after they have developed and tested their programs. In either case, their use requires careful programming in order to avoid undesirable effects due to nondeterminism.

3.4 Buffering, safety and efficiency

When using the high-performance communication primitives `bsp_hpget` and `bsp_hpput` that are not buffered locally, it is important that those data structures are *available* at the end of the superstep. The lifetime of a superstep may span the scope of many procedures. It is therefore quite possible to initiate a communication from a stack allocated data-structure, in a situation where the data structure is out of scope at the end of the superstep. For example, consider the procedure in Figure 3.2 that performs an all-to-all communication. The array `result` is first allocated and then registered at the start of the procedure. A for-loop is then used to copy the value `x` local to each process into the registered arrays `result` in every process at the `bsp_pid`$()^{th}$ position. The *similar* procedure shown in Figure 3.3 is however ill-defined. As the value of `y` communicated by `bsp_hpput` only has scope local to the procedure `do_store` because of the call-by-value semantics of C-procedure invocation, the variable will not be in scope when the communication actually occurs at the second `bsp_sync` within `bad_total_exchange`. This example shows that the user has to be careful when communicating from data structures that do not persist for the lifetime of the program. This is because communication may be delayed until the barrier synchronisation at the end of each superstep.

3.5 Bulk synchronous message passing

In the BSMP system, the primary design decisions were as follows:

```
int *good_total_exchange(int x) {
  int i, *result;
  result=calloc(bsp_nprocs(),
                sizeof(int));

  bsp_register(result);
  bsp_sync();

  for(i=0;i<bsp_nprocs();i++) {
    bsp_hpput(i,
              &x,
              result,
              bsp_pid()*sizeof(int),
              sizeof(int));
  }
  bsp_sync();
  return result;
}
```

Figure 3.2: Well-defined *BSPlib* program

```
int *bad_total_exchange(int x) {
  int i, *result;
  result=calloc(bsp_nprocs(),
                sizeof(int));

  bsp_register(result);
  bsp_sync();

  for(i=0;i<bsp_nprocs();i++) {
    do_store(i,x,result);
  }
  bsp_sync();
  return result;
}

void do_store(int pid,int y,int *result){
  bsp_hpput(pid,
            &y,
            result,
            bsp_pid()*sizeof(int),
            sizeof(int));
}
```

Figure 3.3: Ill-defined *BSPlib* program

- A message consists of two parts: a fixed-length tag and a variable-length payload.
- The length of the tag can vary between supersteps, but all messages sent in a superstep must have the same tag size.
- The messages are buffered locally: after the tag and payload of a message are sent, the user can safely change the memory that stored the tag and payload.

We address the two-part message decision first. Tags are associated with messages for identification purposes. Users are allowed to select a tag size and format that accommodates their application. The programmer can conveniently label each message to identify the order in which the messages were sent, the source process, the data type of the payload, or whatever information is appropriate for the identification of each message.

It is in general convenient to use tags that are of a different data type than the payload. Since FORTRAN does not support mixed-type data structures, multi-part messages are required. For C, it would not have been necessary to treat a tag as a structure that is separate from the payload of the message. However, even for C the use of a separate tag is advantageous when transporting large arrays.

We allow the tag size to change to allow for greater modularity in BSP programs. That is, each subroutine or even each superstep can set the tag as is appropriate for the information that is communicated during the superstep. The fact that a subroutine may not know the tag size of the calling procedure required that `bsp_set_tag_size` return the old tag size so that a subroutine can reset the tag size upon exit as necessary.

We now discuss some issues concerning implementation. The semantics of the BSMP primitives are classified as *buffered locally/buffered remotely*, but the underlying implementation can vary greatly for different systems. The natural implementation is to buffer messages at the destination. Some implementations may choose to buffer messages at the source as well in order to bundle messages and make better use of network bandwidth. Other implementations will immediately send messages through the network and avoid physical source buffering.

Appendix A

Collective communications

Some message passing systems, such as MPI [9], provide primitives for various specialised communication patterns which arise frequently in message passing programs. These include broadcast, scatter, gather, total exchange, reduction, 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 and get which generate arbitrary and unstructured communication patterns. The library we have described can easily be extended to support such structured communications by adding `bsp_broadcast`, `bsp_combine`, `bsp_scatter`, `bsp_gather`, `bsp_scan`, `bsp_exchange`, etc. as higher level operations. These could be implemented in terms of the core operations, or directly on the architecture if that was more efficient.

Acknowledgements

The work of Jonathan Hill and Bill McColl was 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.

The authors would like to thank Rob Bisseling, Richard Miller, David Skillicorn, Bolek Szymanski and Hong Xie for various discussions on BSP libraries.

Bibliography

[1] T Cheatham, A Fahmy, D C Stefanescu, and L G Valiant. Bulk synchronous parallel computing - a paradigm for transportable software. In *Proc. 28th Hawaii International Conference on System Science*, January 1995.

[2] David E. Culler, A. Dusseau, S. C. Goldstein, Arvind Krishnamurthy, S. Lumetta, Torsten von Eicken, and Kathy Yelick. Parallel programming in Split-C. In *Supercomputing'93*, pages 262–273, November 1993.

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1994.

[4] Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, and Thanasis Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996. (to appear).

[5] Mark W. Goudreau, Kevin Lang, Satish B. Rao, and Thanasis Tsantilas. The Green BSP Library. Technical Report 95–11, University of Central Florida, August, 1995.

[6] Simon Knee. Program development and performance prediction on BSP machines using Opal. Technical Report PRG-TR-18-94, Oxford University Computing Laboratory, August 1994.

[7] W. F. McColl. Scalable computing. In J van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000 in LNCS, pages 46–61. Springer-Verlag, 1995.

[8] W. F. McColl and Q. Miller. The GPL language: Reference manual. Technical report, ESPRIT GEPPCOM Project, Oxford university Computing Laboratory, October 1995.

[9] Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society, 1993.

[10] Richard 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, December 1993.

[11] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

Index