BSPlib

The BSP Programming Library

Jonathan M. D. Hill[1]
Bill McColl[1]
Dan C. Stefanescu[2,3]
Mark W. Goudreau[4]
Kevin Lang[5]
Satish B. Rao[5]
Torsten Suel[6]
Thanasis Tsantilas[7]
Rob Bisseling[8]

[1]University of Oxford
[2]Harvard University
[3]Suffolk University
[4]University of Central Florida
[5]NEC Research Institute, Princeton
[6]Bell Laboratories, Lucent Technologies
[7]Columbia University
[8]Utrecht University

`http://www.bsp-worldwide.org/`

May 1997

ANSI C Examples

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Since the earliest days of computing it has been clear that, sooner or later, sequential computing would be superseded by parallel computing.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 [7, 3, 6, 2, 1, 5].

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 the following sections we describe *BSPlib*, a small communications library for programming in an SPMD (Single Program Multiple Data) manner. The main features of *BSPlib* are two modes of communication, one capturing a BSP oriented message passing approach, and the other reflecting a one-sided direct remote memory access (DRMA) paradigm. A higher level library, outlined 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.

Library Contents

Table of Examples

Initialisation

Like many other communications libraries, *BSPlib* adopts a Single Program Multiple Data (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.

Starting and finishing SPMD code

In C

```
void bsp_begin(int maxprocs);

void bsp_end(void)
```

In Fortran

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

SUBROUTINE bspend()
```

Parameters

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

Explanation

Processes are created in a *BSPlib* program by the operations `bsp_begin`[1] 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` pair 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 Page 5 for details.

[1] as a naming convention, C procedures have underscores within them, whereas in FORTRAN we use the same name without any underscores.

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".

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

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 users terminal in a non-deterministic manner. All other types of I/O (e.g., user input and file access) are only guaranteed to work correctly if performed by *process zero*. Therefore this example prints p strings in an arbitrary order.

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.
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.

Simulating dynamic processes

In C

```
void bsp_init(void(*spmdproc)(void),int argc,char **argv)
```

In Fortran

```
SUBROUTINE bspinit(spmdproc)
   INTERFACE
      SUBROUTINE spmdproc
   END INTERFACE
```

Parameters

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

Explanation

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[2] 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.

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[]) {
```

[2]almost all distributed memory machines, e.g. IBM SP2, Cray T3E, Meiko CS-2, Parsytec GC, Hitachi SR2001.

```
  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”.

Halt

The following function provides a simple mechanism for raising errors in *BSPlib* programs. A single process in a potentially unique thread of control can halt an entire *BSPlib* program.

One process stops all

In C

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

In Fortran

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

Parameters

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

Explanation

The function 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.

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.

Enquiry

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.

Number of processes

In C

```
int bsp_nprocs(void);
```

In Fortran

```
INTEGER FUNCTION bspnprocs()
```

Explanation

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 $1 \le p \le maxprocs$, 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 $0 \le m \le p - 1$.

Finding my process identifier

In C

```
int bsp_pid(void);
```

In Fortran

```
INTEGER FUNCTION bsppid()
```

Explanation

The function bsp_pid returns the integer m that uniquely identifies the process executing the function.

Local wall-clock time

In C

```
double bsp_time(void);
```

In Fortran

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

Explanation

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 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 on different processes.*

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.

Barrier synchronisation

In C

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

In Fortran

```
SUBROUTINE bspsync()
```

Explanation

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.

Example

Unlike the previous examples, the following program *attempts* (it is not guaranteed[3]) to serialise the printing by ensuring each process prints its output in turn. This is done by performing p iterations, each separated by a barrier synchronisation, where process i prints "Hello BSP Worldwide" during iteration i.

```
void main(void) {
  int i;
  bsp_begin(bsp_nprocs());
    for(i=0; i<bsp_nprocs(); i++) {
      if (bsp_pid()==i)
        printf("Hello BSP Worldwide from process %d of %d\n",
               i,bsp_nprocs());
      fflush(stdout);
      bsp_sync();
    }
  bsp_end();
}
```

[3] there is no way of guaranteeing that output data will be flushed to a file at the end of the superstep.

Direct Remote Memory Access

One way of performing data communication in the BSP model is to use Direct Remote Memory Access (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 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_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 upon the buffering requirements of the communication operations. The right choice of buffering depends upon 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 nondeterministically chosen; it is implementation dependent which one of the many "colliding" communications should be written into the remote memory area.

Registration

In C

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

In Fortran

```
SUBROUTINE bsppushreg(ident,size)
   <TYPE>, intent(IN) ::  ident
   INTEGER, intent(IN)::  size

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

Parameters

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

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

Explanation

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 executes

$$\texttt{bsp\_push\_reg}(\mathit{ident}_i, \mathit{size}_i)$$

then the entry $\langle\langle \mathit{ident}_0, \mathit{size}_0\rangle, \ldots, \langle \mathit{ident}_{p-1}, \mathit{size}_{p-1}\rangle\rangle$ 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,

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

and the registration sequence[4] is $ss \mathbin{+\!\!+} [s] \mathbin{+\!\!+} \overline{ss}$, where entry s is the most recent entry containing tgt_l (i.e., the l^{th} element of s is $\langle \mathit{tgt}_l, n_l\rangle$, and there is no entry $\overline{s}$ in ss such that the l^{th} element of $\overline{s}$ is $\langle \mathit{tgt}_l, m_l\rangle$), 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 $\mathit{tgt}_r + \mathit{offset}$ 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 $\mathit{offset} + \mathit{nbytes} > n_r$.

The effect of the de-registration

$$\texttt{bsp\_pop\_reg}(\mathit{ident}_l)$$

is that given the registration sequence $ss \mathbin{+\!\!+} [\langle\langle \mathit{ident}_0, \mathit{size}_0\rangle, \ldots, \langle \mathit{ident}_{p-1}, \mathit{size}_{p-1}\rangle\rangle] \mathbin{+\!\!+} \overline{ss}$, and suppose that there does not exist an entry $\overline{s}$ in ss such that the l^{th} element of $\overline{s}$ is $\langle ident_l, m_l\rangle$, then the registration sequence is changed to $ss \mathbin{+\!\!+} \overline{ss}$ *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.

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.

[4] the operator $+\!\!+$ is used to concatenate two sequences together.

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[5]. 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[6], 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.

Copy to remote memory

In C

```
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);
```

[5] the array bspunregistered may be used by FORTRAN programmers.
[6] this is not always the case, as some optimising C compilers *un-static* statics.

In Fortran

```
SUBROUTINE bspput(pid,src,dst,offset,nbytes)
   INTEGER, intent(IN)::  pid, offset, nbytes
   <TYPE>, intent(IN) ::  src
   <TYPE>, intent(OUT)::  dst

SUBROUTINE bsphpput(pid,src,dst,offset,nbytes)
   INTEGER, intent(IN)::  pid, offset, nbytes
   <TYPE>, intent(IN) ::  src
   <TYPE>, intent(OUT)::  dst
```

Parameters

`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.

Explanation

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 *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.

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 $\texttt{bsp\_nprocs}() - i - 1$.

```
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 $\texttt{bsp\_nprocs}() - \texttt{bsp\_pid}() - 1$. 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 popregister 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.*

Example

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

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

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[7] with each process owning an $\mathtt{n}/p$ 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,pid,local_idx,n_over_p= n/bsp_nprocs();
  if ((n % bsp_nprocs()) != 0)
    bsp_abort("{put_array} n=%d not divisible by p=%d",
```

[7] i.e., process zero gets elements 0 to $\mathtt{n}/p - 1$, process one gets $\mathtt{n}/p$ to $2\mathtt{n}/p - 1$, etc.

```
              n,bsp_nprocs());
  bsp_push_reg(xs,n_over_p*sizeof(int));
  bsp_sync();

  for(i=0;i<n_over_p;i++) {
    pid       = xs[i]/n_over_p;
    local_idx = xs[i]%n_over_p;
    bsp_put(pid,&xs[i],xs,local_idx*sizeof(int),sizeof(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 to `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 `pid` is the same as `bsp_pid()`, then a local assignment (i.e., memory copy) will occur *at the end of the superstep*.

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 `bsp_put(pid, src, 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 `pid` = `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, even for the FORTRAN bindings of the operations.

Copy from remote memory

In C

```
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);
```

In Fortran

```
SUBROUTINE bspget(pid,src,offset,dst,nbytes)
   INTEGER, intent(IN)::  pid, offset, nbytes
   <TYPE> intent(IN) ::  src
   <TYPE> intent(OUT) ::  dst

SUBROUTINE bsphpget(pid,src,offset,dst,nbytes)
   INTEGER, intent(IN)::  pid, offset, nbytes
   <TYPE> intent(IN) ::  src
   <TYPE> intent(OUT) ::  dst
```

Parameters

`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.

Explanation

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.

Example

The procedure `get_array` is the dual of `put_array` defined earlier. The procedure is semantically equivalent to the concurrent assignment:

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

```
void get_array(int *xs, int n) {
  int i,pid,local_idx,n_over_p=n/bsp_nprocs();
  if (n % bsp_nprocs())
    bsp_abort("{get_array} %d not divisible by %d",
              n,bsp_nprocs());
  bsp_push_reg(xs,n_over_p*sizeof(int));
  bsp_sync();

  for(i=0;i<n_over_p;i++) {
    pid       = xs[i]/n_over_p;
    local_idx = xs[i]%n_over_p;
    bsp_get(pid,xs,local_idx*sizeof(int),&xs[i],sizeof(int));
  }
  bsp_sync();
  bsp_pop_reg(xs);
}
```

In this example buffering is necessary as processes need to read data before it is overwritten. Thus, for a given array element `xs[i]`, all reads generated by `bsp_gets` are performed ahead of the writes generated by any buffered operation, including those due to the `bsp_get` of the process on which `xs[i]` resides.

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 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, int nelem) {
  int *local_sums,i,j,result=0;
  for(j=0;j<nelem;j++) result += xs[j];
  bsp_push_reg(&result,sizeof(int));
  bsp_sync();

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

  result=0;
  for(i=0;i<bsp_nprocs();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.

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.

Bulk Synchronous Message Passing

Direct 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 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 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, 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.

Choose tag size

In C

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

In Fortran

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

Parameters

`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.

Explanation

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; it then becomes *valid*.

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, then 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.

Send to remote queue

In C

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

In Fortran

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

Parameters

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.

Explanation

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.

Notes

1. The size of the tag used in bsp_send will depend upon either the size of tag that was active 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.

Number of messages in queue

In C

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

In Fortran

```
SUBROUTINE bspqsize(nmessages,accum_nbytes)
   INTEGER, intent(OUT) ::  nmessages,accum_nbytes
```

Parameters

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

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

Explanation

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.

Notes

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

Getting the tag of a message

In C

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

In Fortran

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

Parameters

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.

Explanation

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 upon the value set by bsp_set_tagsize.

Move from queue

In C

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

In Fortran

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

Parameters

`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`.

Explanation

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

Example

In the algorithm shown below, an n element vector distributed into n/p chunks on p processes undergoes a communication whereby all the nonzero 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,
      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<bsp_nprocs();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 nonzero 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 nonzero 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.

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.

A lean method for receiving a message

In C

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

Parameters

`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).

Explanation

The operation `bsp_hpmove` is a non-copying method of receiving messages that is available in languages with pointers such as C, but not standard 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, as it requires the availability of sufficient contiguous memory. The storage referenced by these pointers remains valid until the end of the current superstep.

A Collective communications

Some message passing systems, such as MPI [4], 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_bcast`, `bsp_fold`, `bsp_scatter`, `bsp_gather`, `bsp_scan`, `bsp_exchange`, etc. as higher level operations. These will be implemented in terms of the core operations, or directly on the architecture if that was more efficient. For modularity and safety, all collective communications will have an implicit registration, *within the routine*, of any arguments that are required to be communicated.

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 Paul Crumpton, Alex Gerbessiotis, Tony Hoare, Antoine Le Hyaric, Tim Lanfear, Bob McLatchie, Richard Miller, David Skillicorn, Bolek Szymanski and Hong Xie for various discussions on BSP libraries. Thanks also to Jim Davies for improving the typesetting in TEX.

References

[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*, volume II, pages 268–275. IEEE Computer Society Press, January 1995.

[2] 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*, pages 1–12, June 1996.

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

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

[5] 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.

[6] David Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, to appear 1997.

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