The *BSPlib* library

(Adapted from presentation

by Jonathan Hill and Paul Crumpton)

What is *BSPlib*, and what is its relation to BSP

Overview

- What is the BSP model?
- What is *BSPlib*, and what is it's relation to BSP?
- Data-parallelism in a Single Program Multiple Data (SPMD) programming style.
- Comparisons to MPI/PVM: why is *BSPlib* needed?
- Direct Remote Memory Access (DRMA)
  - put and get
  - buffered vs. unbuffered communication
  - registration: why is it useful?
- Bulk Synchronous Message Passing (BSMP)
- Where is *BSPlib* and how do I install it?

The Bulk Synchronous Parallel model

BSP is a *model* of parallel computation. It describes, in general terms, how a parallel computation should proceed.

BSP does not prescribe the way in which:

- local computations are carried out
- communication actions are expressed.

The BSP model (cont.)

The essence of the BSP model is the notion of a *superstep*. A superstep is decomposed into three phases:

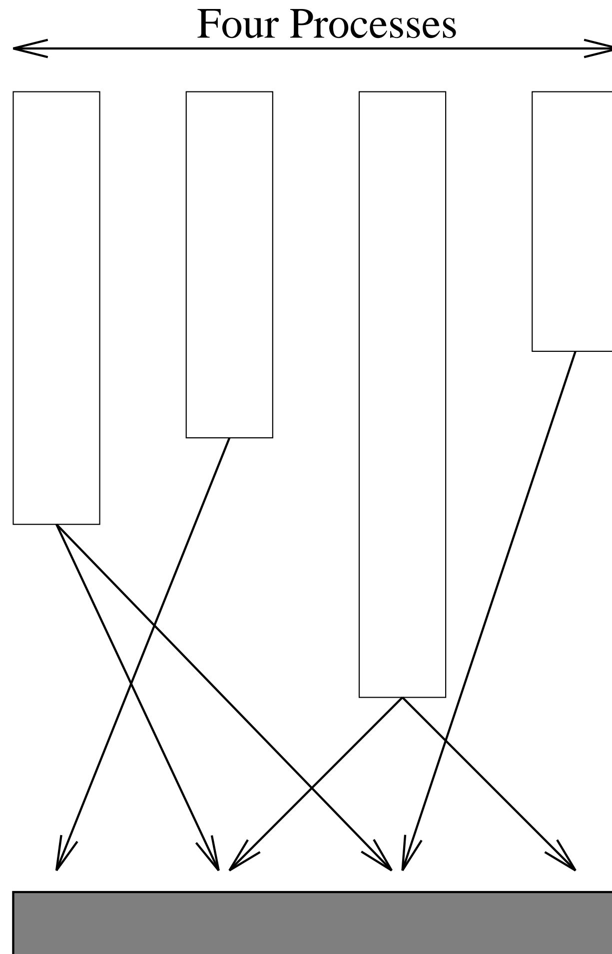
Local computation within each process on local data-values.

Non-blocking communication can be initiated with other processors.

A barrier synchronisation of *all processes* is used to delimit the end of the superstep where all communications are guaranteed to have occurred by.

Communication and synchronisation are completely decoupled

The BSP model (cont.)



Local Computation Occuring
in parallel among four processes

Global communication

Barrier synchronisation

What is *BSPlib*?

A standard library for BSP programming developed under the auspices of BSP World Wide by:

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

Authors' affiliations:
(1) University of Central Florida
(2) Oxford University
(3) NEC Research Institute, Princeton
(4) AT&T Bell Labs, NJ.
(5) Harvard University
(6) Columbia University

see `http://www.bsp-worldwide.org` *for more details*

BSPlib (cont.)

A small library of 19 operations (MPI-1: 129 operations; MPI-2: 202 operations, PVM: 90 operations):

Class	Operation	Meaning
Initialisation	`bsp_init`	Simulate dynamic processes
	`bsp_begin`	Start of SPMD code
	`bsp_end`	End of SPMD code
Enquiry	`bsp_pid`	Find my process id
	`bsp_nprocs`	Number of threads
	`bsp_time`	Local time
Synchronisation	`bsp_sync`	Barrier synchronisation
DRMA	`bsp_pushregister`	Make region globally visible
	`bsp_popregister`	remove global visibility
	`bsp_put`	Push to remote memory
	`bsp_get`	Pull from remote memory
BSMP	`bsp_set_tag_size`	Choose tage size
	`bsp_send`	Send to remote queue
	`bsp_get_tag`	Match tag with message
	`bsp_move`	Fetch from queue
Halt	`bsp_abort`	One process halts all
High Performance	`bsp_hpput`	Unbuffered versions
	`bsp_hpget`	of communication
	`bsp_hpmove`	primitives

Data-parallelism in a SPMD programming style

Single **P**rogram **M**ultiple **D**ata:

- multiple copies (processes) of the same program running in parallel.
- each process has its own local data-structures.

SPMD programming (cont..)

Generally, given a problem of size N, a parallel solution divides the problem into N/p chunks, and solves each of the p chunks in parallel.

The problem of parallelisation is then to arrange phases of communication and local computation, such that any data required for the local computation within a superstep is communicated in the prior superstep.

SPMD programming (cont..)

```
void main() {
  int x;
  bsp_begin(3);
    x=bsp_pid();
    if (x%2) printf("I am an odd process: %d\n",x);
    else     printf("I am an even process: %d\n",x);
  bsp_end();
}
```

```
pine.comla> ./foo
I am an odd process: 1
I am an even process: 0
I am an even process: 2
```

DRMA: Direct Remote Memory Access

- One-sided communication—a process can copy a block of data from its own memory, into the memory of another process, *without the active participation of the remote process*.
- Two different styles:

get that *reads* or fetches a copy of a data-structure from a remote process into the process initiating the get.

put offloads or *writes* data from the initiating process into a remote process.

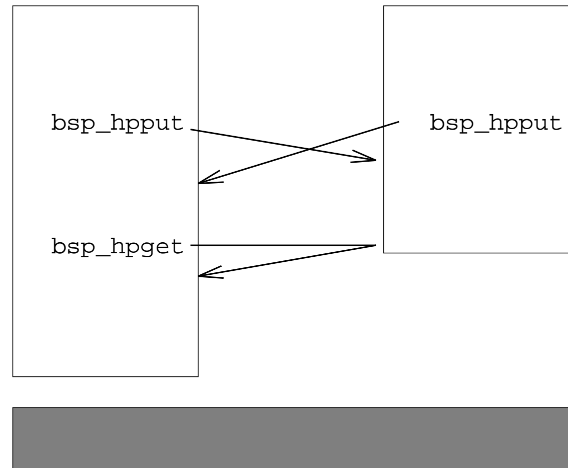
DRMA: Direct Remote Memory Access (cont..)

The semantics (i.e., meaning) of the DRMA operations depend upon the interpretation of a superstep. i.e., do the communications:

1. **buffered**: take effect *at* the end of the superstep.
2. `high performance unbuffered`: take effect *any time up to* the end of the superstep.

These two different styles of DRMA are provided in *BSPlib*.

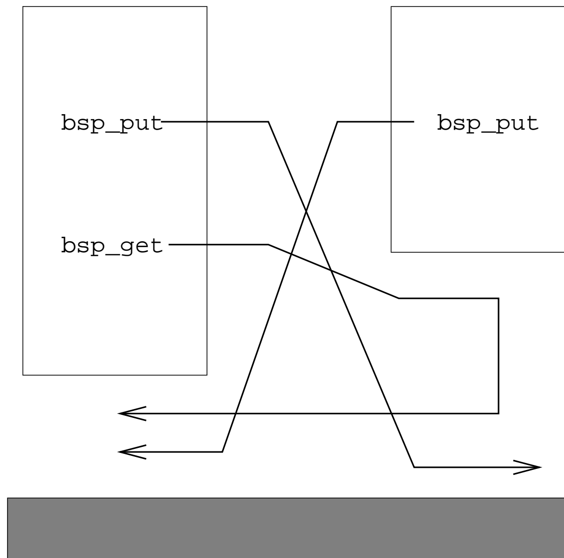
BSPlib: high-performance DRMA



Communication and computation can be overlapped. However,

- data-structures communicated can not be changed locally until the end of the superstep.
- hard to reason about programs due to the non-determinism of:
  - DRMA writes changing local variables
  - interaction of puts and gets

BSPlib: standard (aka buffered) DRMA



The phases of local computation, communication, and synchronisation do not overlap. Therefore,

- communicated data-structures may be changed locally after this kind of DRMA operation
- the only non-determinism is due to multiple processes writing into the same location(s).

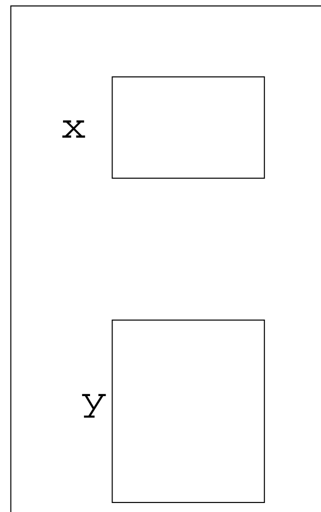
DRMA: registration, the problem

When using DRMA, if process A executes:

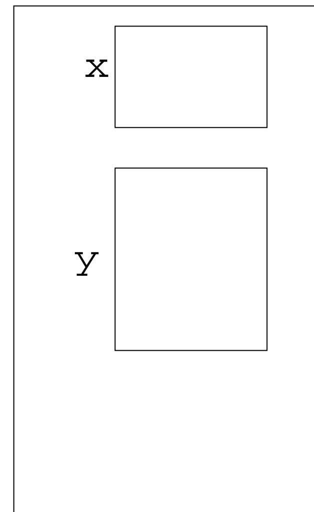
$$\texttt{bsp_put}(B, x, y, 0, \texttt{sizeof}(x));$$

then the data-structure x on process A is moved into the data-structure y on process B, where x and y are the *names* of arrays in the user program.

- As *BSPlib* is a SPMD library, then x and y will probably exist on all the processes.
- At runtime, on a particular process A, x and y reference the data-structures denoted by x and y on process A.
- All the different values of x, in all the processes will probably be different.



Memory on A



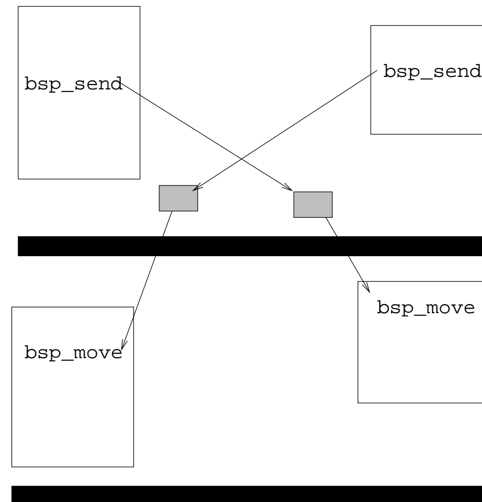
Memory on B

Registration

Registration is the process by which the user informs *BSPlib* the names of the *same* data-structure on each process. It enables:

- Communication to and from heap allocated data
- Communication to and from stack allocated data
- *BSPlib* to work in heterogeneous environments
- bounds checking of communications

BSPlib: Bulk Synchronous Message Passing



- Standard non-blocking send and receive operations.
- all messages have a tag and payload field. The tag is of fixed size for all messages in a superstep. The payload can be arbitrarily sized.
- Synchronisation is guaranteed by the *barrier at the end of the superstep*.

Choosing between DRMA and BSMP

- DRMA should be used in problems in which each process knows of the existence, *and size* of remote areas of memory.
- BSMP is better suited to irregular problems where the recipient of data only knows where any incoming data should be stored.

Similarities between *BSPlib*, PVM and MPI

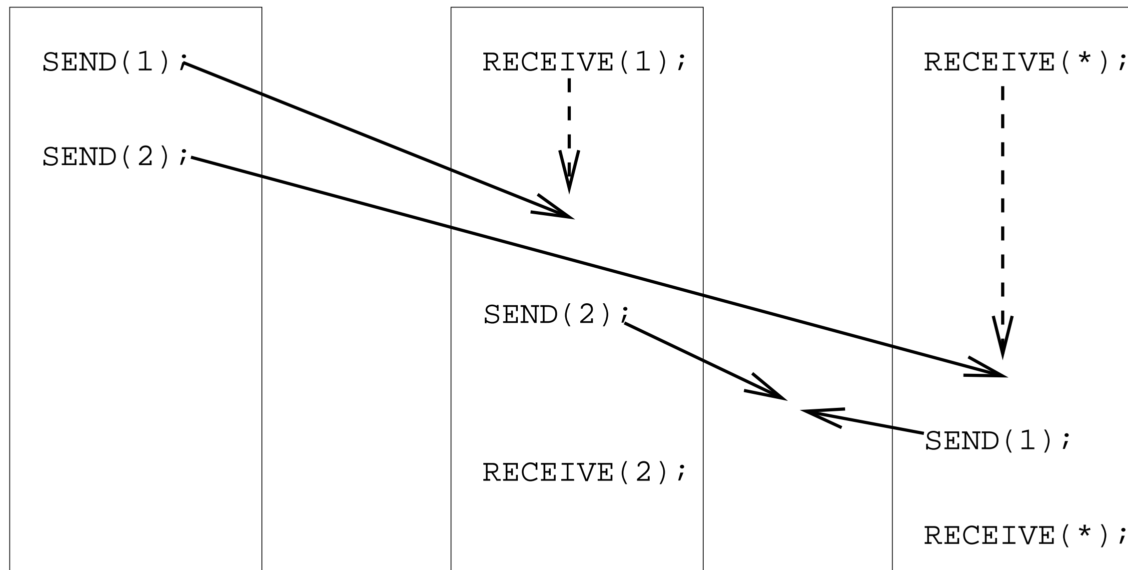
- All can be used in a SPMD programming mode.
- Portable
- Implemented on a wide-variety of machines.

Comparisons to *synchronous* message passing (e.g., part of PVM/MPI)

Pairwise, synchronous message passing can suffer from:

- Deadlock
- Complexity of send/receive actions—spaghetti communication code.
- No *simple* analytic cost model for performance prediction
- Generally no global consistent state, therefore hard to debug.
- On shared or distributed memory architectures with powerful global communication, can be less efficient than BSP.

Example of spaghetti-type communication and deadlock



Comparisons to *asynchronous* message passing (e.g., part of PVM/MPI)

Asynchronous message passing in conjunction with barriers can be used to program in a BSP style.

However, message passing systems are not often tuned for this way of working—e.g., the MPI barrier synchronisation on the Power Challenge is 32 times slower than *BSPlib*'s barrier.

In summary, BSP offers:

BSP offers:

- a simple programming discipline based on supersteps that makes it easier to determine the correctness of programs;
- a simple cost model for performance analysis and prediction;
- efficient implementations on many machines.
- small library
- no deadlock!

However, it only works on systems with the same byte ordering.

Availability of the BSPlib software

From `http://www.BSP-Worldwide.org/implmnts/oxtool.htm`

1. Silicon Graphics Power Challenge (Irix 6.x)
2. IBM SP2
3. Cray T3D
4. Parsytec Explorer
5. Convex SPP
6. Digital 8400 and Digital Alpha farm
7. Hitachi SR2001

Plus generic versions on top of either MPI or system V shared memory.

How to install *BSPlib*

Requirements:

1. Perl
2. C, C++,or a F77 compiler.
3. Tcl/Tk if call-graph profiling is required

How to install *BSPlib* (cont.)

For a self-contained, uni-processor implementation of the library where the distribution has been untarred into `/users/fred/BSPlib/`:

```
cd /users/fred/BSPlib/
env BSP_ONLINE_PROCS=1 configure --prefix=/users/fred/BSPlib/
make
```

How to install *BSPlib* (cont.)

For an installation within the `/usr/local/` hierarchy, for a 4-processor machine where the distribution has been untarred into `/users/fred/BSPlib/`:

```
cd /users/fred/BSPlib/
env BSP_ONLINE_PROCS=4 configure --prefix=/usr/local/
make install
```

BSPlib: writing code

Overview

All library routines are described by examples, for details of arguments see BSPlib man handout.

Compiling a BSPlib program

BSPlib: core

- control and enquiry
- communication (DRMA)
- communication (message passing)

BSPlib: level 1 (these call BSPlib core)

- broadcast, reduction (fold) etc...

Compiling a BSPlib program

The scripts `bspcc` and `bspf77` compile and link FORTRAN and C programs respectively, eg:

```
bspf77 prog1.f
bspcc prog1.c
```

(NB. The scripts link required libraries, and initiate any special compilers. For more info type "`bspcc -help`" or man page)

The number of processes to be initiated can be set by the environment variable `BSP_PROCS` eg:

```
setenv BSP_PROCS 17
```

NB (1) can be machine dependent; (2) can do it by argument

Control and Enquiry

`bsp_begin(maxprocs)` start of SPMD with at most `maxprocs` processes (usually first line of main)

`bsp_end()` end of SPMD (usually last line of main)

`bsp_nprocs()` if called after `bsp_begin` it returns the number of processes currently running, if called before `bsp_begin` it returns the number of processes asked for.

`bsp_pid()` returns process id where $0 \leq$ `bsp_pid()` $\leq$ `bsp_nprocs()-1`

`bsp_time()` returns elapsed *wall-clock* time

`bsp_abort(...)` abnormal exit

(NB for FORTRAN procedure names, remove underscores_)

Program Layout C

```
#include <stdio.h>
#include "bsp.h"
void main(void){
  bsp_begin(bsp_nprocs());
        ....
        ....
        ....
        ....
  bsp_end();
}

%bspcc prog.c
%setenv BSP_PROCS 8
%a.out
```

Program Layout FORTRAN

```
 implicit none
 include 'fbsp.h'
 call bspbegin(bspnprocs())
     .....
     .....
     .....
     .....
call bspend()
end
```

```
%bspf77 prog.f
%setenv BSP_PROCS 8
%a.out
```

Prog?.?

All the example programs in this lecture are given in C (ANSI) and FORTRAN. They are lacking in comments (so they fit on the slide) and are totally trivial.

Prog1: writes `bsp_pid` and `bsp_nprocs` to standard output (the terminal).

Prog1.c

```
#include <stdio.h>
#include "bsp.h"
void main(void){
  bsp_begin(bsp_nprocs());
  printf(" my id is = %d, out of %d processes \n",
          bsp_pid(),bsp_nprocs());
  bsp_end();
}
```

Prog1.f

```
      implicit none
      include 'fbsp.h'
c
      call bspbegin(bspnprocs())
c
      write(*,*) 'my id is = ',bsppid(),' out of',
     &     bspnprocs(),' processes'
c
      call bspend()
      end
```

running Prog1.c

```
bspcc prog1.c
setenv BSP_PROCS 8
a.out
 my id is = 5, out of 8 processes
 my id is = 6, out of 8 processes
 my id is = 7, out of 8 processes
 my id is = 1, out of 8 processes
 my id is = 2, out of 8 processes
 my id is = 3, out of 8 processes
 my id is = 4, out of 8 processes
 my id is = 0, out of 8 processes
```

Note

- The number of processes is set by the environment variable `BSP_PROCS` (NB this can be set other ways using `bsp_init()`).
- All standard output is handled naturally, however, the order it appears is random between synchronizations.
- Here we are only interested in the SPMD fragments of code between `bsp_begin()` and `bsp_end()`.

DRMA-Direct Remote Memory Access

We recommended DRMA style whenever convenient.

`bsp_pushregister(...)` register a piece of memory on a stack as read/writable for the put and get routines
(NB. takes effect after next sync)

`bsp_popregister(...)` pop a registration from the stack (NB. can only unregister the last thing you registered).

`bsp_put(...)` put some local *source* into a *destination* on a remote processor, this remote space *must* be registered.

`bsp_get(...)` get some remote *source* and copy it into a local *destination*, this remote source *must* be registered.

`bsp_sync()` all processors synchronize, and all pending puts, gets and messages are completed after this call.

DRMA-Direct Remote Memory Access

The general use of DRMA communication is:

1. all processors *MUST* register the space into which remote "reads" and "writes" will occur;
2. calls to `bsp_put` (as many as you like);
3. calls to `bsp_get` (as many as you like);
4. call to `bsp_sync` which denotes the end of the super-step, all processors barrier synchronize, all communication is completed after this call.

DRMA-Direct Remote Memory Access

Arguments of puts and gets have (**source**, **destination**), where

source is to be "read"

destination is to be "written"

	source	destination	must register	"read"	"write"
put	local	remote	destination	at call	at sync
get	remote	local	source	at sync	at sync

with `bsp_put()` and `bsp_get()` ALL communication occurs at `bsp_sync()`

Prog2 and 3

Prog2 All processors write their pid into ARRAY(pid) on process 0 using puts.

Prog3 All processors write their pid into ARRAY(pid) on process 0 using gets.

fragment of Prog2.c (use of `bsp_put(...)`)

```
int ARRAY[10]={0},i;
bsp_begin(bsp_nprocs());

bsp_pushregister(ARRAY, sizeof(ARRAY));
bsp_sync();
i = bsp_pid();
bsp_put(0,                          /* id of destination */
        &i,                         /* source(read)*/
        ARRAY,                      /* destination (written) */
        bsp_pid()*sizeof(int),      /* offset */
        sizeof(int));               /* size of put */
bsp_sync();

printf(" pid = %d,  ARRAY =",bsp_pid());
for(i=0; i<bsp_nprocs(); i++) printf(" %d",ARRAY[i]);
printf("\n");
bsp_end();
```

fragment Prog2.f (use of `call bspput(...)`)

```
      include 'fbsp.h'
      integer ARRAY(10),i
c
      call bspbegin(bspnprocs())
      call bsppushregister(ARRAY, 10*BSPINT)
      call bspsync()
      call bspput(0,                 !pid of put
     &            bsppid(),          !source(read)
     &            ARRAY,             !destination(written)
     &            bsppid()*BSPINT,!offset
     &            BSPINT)            !size(bytes)
      call bspsync
c
      write(*,'(a,i2,a,99i2)') ' pid =',bsppid(),' ARRAY=',
     &     (array(i),i=1,bspnprocs())
      call bspend()
      end
```

running Prog2

```
bspf77 prog1.f
setenv BSP_PROCS 8
a.out
 pid = 0 ARRAY= 0 1 2 3 4 5 6 7
 pid = 6 ARRAY= 0 0 0 0 0 0 0 0
 pid = 7 ARRAY= 0 0 0 0 0 0 0 0
 pid = 1 ARRAY= 0 0 0 0 0 0 0 0
 pid = 5 ARRAY= 0 0 0 0 0 0 0 0
 pid = 3 ARRAY= 0 0 0 0 0 0 0 0
 pid = 2 ARRAY= 0 0 0 0 0 0 0 0
 pid = 4 ARRAY= 0 0 0 0 0 0 0 0
```

running Prog2 with more processors

If BSP_PROCS > 10 then some processors will try to access off the end of `ARRAY`

```
bspf77 prog1.f
setenv BSP_PROCS 11
a.out
ABORT(pid  0):{bsp_sync} line 12 of "prog2.f"
        Out of bound communication (4 bytes at offset 40)
        was attempted into a data-structure that was only
        registered with 40 bytes at line 6 of "prog2.f"
```

The default options of `bspf77` have checking enabled (this includes not registering space). All this checking makes the communication performance poor (see `bspf77 -help` for options).

fragment of Prog3.c (use of bsp_get)

```
int ARRAY[10]={0},i,my_id;
bsp_begin(bsp_nprocs());

my_id = bsp_pid();
bsp_pushregister(&my_id, sizeof(my_id));
bsp_sync();
if(bsp_pid()==0)
  for(i=0; i<bsp_nprocs(); i++) {
    bsp_get(i,                      /* id of destination */
            &my_id,                 /* source (read) */
            0,                      /* offset */
            &ARRAY[i],              /* destination(written) */
            sizeof(int));           /* size of put */
  }
bsp_sync();    /* sync not in if loop */
printf(" pid = %d,  ARRAY =",bsp_pid());
for(i=0; i<bsp_nprocs(); i++) printf(" %d",ARRAY[i]);
```

fragment of Prog3.f (use of `call bspget`)

```
      my_id = bsppid()
      call bsppushregister(my_id, BSPINT)
      call bspsync()
      if(bsppid() .eq. 0) then
         do i=0,bspnprocs()-1
            call bspget(i,          !pid of get
     &                  my_id,      !source(read)
     &                  0,          !offset
     &                  ARRAY(i),   !destination (written)
     &                  BSPINT)     !size(bytes)
         enddo
      endif
      call bspsync()  ! sync not in if loop
c
      write(*,'(a,i2,a,99i2)') ' pid =',bsppid(),' ARRAY=',
     &     (array(i),i=1,bspnprocs())
      call bspend()
```

Prog4.c what will happen ?

```
#include <stdio.h>
#include "bsp.h"
void main(void){
  int i;
  bsp_begin(bsp_nprocs());

  bsp_pushregister(&i,  sizeof(int));
  bsp_sync();
  for(i=0; i<bsp_nprocs(); i++){
    if(bsp_pid()==0) bsp_put(i, &i, &i, 0, sizeof(int));
  }
  bsp_sync();
  if(i != bsp_pid())
    bsp_abort(" something is wrong i=%d\n",i);
  bsp_end();
}
```

Prog4.f what will happen ?

```
      implicit none
      include 'fbsp.h'
      integer i
c
      call BspBegin(BspNprocs())
      call BspPushRegister(i, BSPINT)
      call BspSync()
c
      do i=0,BspNprocs()-1
         if(BspPid().eq.0) call BspPut(i,i,i,0,BSPINT)
      enddo
      call BspSync()
      if(i.ne.BspPid()) call BspAbort(" something is wrong")
c
      call BspEnd()
      end
```

Prog4.f what will happen ?

Nothing as the put “reads” the source `i` into an internal buffer at the call, and the destination `i` is “written” to at the sync.

This type of “side–effect” (ie space being over–written at a call that does not reference that space) is common in C programs, NOT for FORTRAN.

Prog5.c what will happen ?

```
#include "bsp.h"
void main(void){
  int left_id;
  bsp_begin(bsp_nprocs());

  bsp_pushregister(&left_id,  sizeof(int));
  bsp_sync();

  /* get id of "left" processors */
  left_id= (bsp_pid()+1)%bsp_nprocs();
  bsp_get(left_id, &left_id, 0, &left_id, sizeof(int));
  left_id = bsp_pid();
  bsp_sync();

  if((bsp_pid()+1)%bsp_nprocs() != left_id)
    bsp_abort(" something is wrong \n");
  bsp_end();
```

Prog5.f what will happen ?

```
      implicit none
      include 'fbsp.h'
      integer left_id
c
      call BspBegin(BspNprocs())
      call BspPushRegister(left_id, BSPINT)
      call BspSync()
c
      left_id = mod(BspPid()+1, BspNprocs())
      call BspGet(left_id,left_id,0,left_id,BSPINT)
      left_id = BspPid()
      call BspSync()
c
      if(mod(BspPid()+1, BspNprocs()).ne.left_id)
     &        call abort(" something is wrong")
c
      call BspEnd()
```

Prog5.f what will happen ?

Nothing as the get “reads” the source `left_id` into an internal buffer at the `call bspsync()` when it contains the pid of the current processor. This then gets overwritten after the communication in the `call bspsync()` with the value of `left_id` in the left pid.

This is a most confusing bit of code!

Notes on Puts and Gets

- On a call to `bsp_put()` the source is “read” at the call and the destination is “written” at the next `bsp_sync()`.
- On a call to `bsp_get()` the source is “read” and the destination “written” at the next `bsp_sync()`.
- Doing puts and gets into oneself is perfectly OK.
- Registration of remote “read” or “write” is essential.
- Non buffered puts and gets exist (`bsp_hpget` and `bsp_hpput`) where communication can happen anywhere.
- This buffering ensures that the communication outcome is pretty well determined (but not completely).
- All `bsp_get()`’s are done before `bsp_put()`’s at the sync.

BSMP-Bulk Synchronous Message Passing

Sometimes the destination is unknown to the sending process, consequently message passing is required.

`bsp_set_tag_size(size)` each message has a tag of `size` bytes.

`bsp_send(...)` transmits an arbitrary sized payload and fixed size tag, that can be interrogated by receiver after the next sync.

`bsp_get_tag(...)` gets the tag and status of a received messages.

`bsp_move(...)` moves a received message into user space.

(NB: All communication occurs at the sync)

(NB: no registration of payload is required)

Prog6.c message passing

```
#include <stdio.h>
#include "bsp.h"
void main(void){
  int tag_size,tag,i,status,received,message1=9,message2=8;
  bsp_begin(bsp_nprocs());

  tag_size = sizeof(int);
  bsp_set_tag_size(&tag_size); /* tag will be an integer */

  /* pid=0 send message1 and message2 to all processors */
  if(bsp_pid() == 0){
    for(i=0; i<bsp_nprocs(); i++){
      bsp_send(i, (tag=45,&tag), &message1, sizeof(int));
      bsp_send(i, (tag=50,&tag), &message2, sizeof(int));
    }
  }
  bsp_sync(); /* all messages are sent here */
```

```
  /* process all messages received */
  while( (bsp_get_tag(&status, &tag),status) != -1 ){
    if(tag == 45){
      bsp_move(&received, sizeof(int));
      if(received != message1)   bsp_abort(" error 1\n");
    } else if(tag == 50){
      bsp_move(&received, sizeof(int));
      if(received != message2)   bsp_abort(" error 2\n");
    } else {
      bsp_abort(" pid = %d got unknown tag=%d\n",bsp_pid(),tag);
    }
  }
}
```

Prog6.f message passing

```
      include 'fbsp.h'
      integer tag_size,tag,i,status,received,message1,message2
      data message1,message2 /9,8/
c
      call BspBegin(BspNprocs())
      call BspSetTagSize(BSPINT) !tag is a integer
c
c     pid=0 send message1 and message2 to all processors
c     ----------------------------------------------------
      if(BspPid().eq.0) then
         do i=0,BspNprocs()-1
            call BspSend(i, 45, message1, BSPINT)
            call BspSend(i, 50, message2, BSPINT)
         enddo
      endif
      call BspSync()  ! all messages are sent here
```

```
c
c     process all messages received
c     ----------------------------
 100  continue
      call BspGetTag(status, tag)
      if(status.ne.-1) then
         if(tag.eq.45) then
            call BspMove(received, BSPINT)
            if(received.eq.message1) call BspAbort("error1")
         else if(tag.eq.50) then
            call BspMove(received, BSPINT)
            if(received.eq.message2) call BspAbort("error2")
         else
            call BspAbort(" receiver unknown")
         endif
         goto 100
      endif
```

Notes on message passing

- Need the “clumsy” code to enable to process messages with any tag.
- Need to be able to do something with each message as it’s looked at.
- Can be very useful if the destination of a message in not known by local process.
- We advise use of this only when “stuck”.
- All communication occurs at the sync.
- No registration of the payload is required.
- All received messages are “trashed” at the next sync.

BSPlib: level-1

All these routines are implemented using the core routines; and are thus for convenience. Two very useful ones are:

`bsp_bcast(...)` broadcasts data from one processor to all others.
(need to register the source)

`bsp_fold(...)` performs a reduction operation (ed sum, max, min over all processors).
(don't need to register the source)

Prog7.c what will happen ?

```
void max(int *res, int *l, int *r, int b){*res =(*l>*r)?*l:*r;}

void main(void){
  int reduce,srce,dest,max_pid;
  bsp_begin(bsp_nprocs());

  /* get max_pid*/
  srce = bsp_pid();
  bsp_fold(max, &srce, &max_pid, sizeof(int));

  /* broadcast from max_pid */
  bsp_pushregister(&dest, sizeof(int));
  bsp_sync();
  bsp_bcast(max_pid, &srce, &dest, sizeof(int));

  printf(" pid=%d, dest=%d\n", bsp_pid(),dest);
  bsp_end();
```

Prog7.f what will happen ?

```
      integer max_pid,dest
      external maxI
c
      call BspBegin(BspNprocs())
      call BspFold(maxI, BspPid(), max_pid, BSPINT)
c
      call BspPushregister(dest, BSPINT)
      call BspSync()
      call BspBcast(max_pid, BspPid(), dest, BSPINT)
c
      write(*,*) 'pid=',BspPid(),'dest=',dest
      call BspEnd()
      end
c
      subroutine maxI(res,l,r,b)
      integer res,l,r,b
      res=max(l,r)
```

running Prog7.c

```
bspcc prog7.c
setenv BSP_PROCS 10
a.out
 pid=0, dest=9
 pid=9, dest=9
 pid=3, dest=9
 pid=1, dest=9
 pid=4, dest=9
 pid=6, dest=9
 pid=7, dest=9
 pid=8, dest=9
 pid=2, dest=9
 pid=5, dest=9
```

Tips using BSPlib

1. data used in puts and gets must be registered !
2. all processes must register at the same point !
3. all processes must reduce at the same point !
4. do not put syncs in logic determined by pid !
5. use `bsp_sync()`'s freely when debugging !
6. remember to set `BSP_PROCS`
7. remember to compile (not just link) using `bspcc` `bspf77`
8. good luck !