P U B - Library

Paderborn **U**niversity **B**SP

Release 6.0

Draft: User Guide
and
Function Reference

http://www.uni-paderborn.de/~pub
pub@uni-paderborn.de

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

[1]Heinz Nixdorf Institute and Department of Computer Science
University of Paderborn
Fuerstenallee 11, 33102 Paderborn, Germany
{bono,ivo,inri}@uni-paderborn.de

[2]Laboratory of Computer Architecture and Digital Technique
Faculty of Information Technology and Systems
Delft Technical University
Mekelweg 4, 2628 CD Delft, The Netherlands
B.H.H.Juurlink@its.tudelft.nl

Contents

1 User Guide **3**

1.1 Introduction . . . 3

1.2 Environment . . . 3

1.3 Programming with PUB . . . 5

1.4 Simulator . . . 11

1.5 BSP Worldwide Standard . . . 11

2 Reference **12**

2.1 List of functions . . . 12

2.2 Description of functions . . . 14

1 User Guide

1.1 Introduction

The PUB–Library (Paderborn University BSP–Library) is a C–library of communication routines. These routines allow implementing algorithms which are designed for the BSP model (bulk synchrounus parallel model). The BSP model divides algorithms into several parts called supersteps. In each superstep a processor can work on local data and send messages. At the end of the superstep a barrier synchronization takes place and all processors receive the messages which were sent in the previous superstep. Further informations concerning the BSP model can be received from [Val90].

PUB offers functions for both message passing and remote memory access. Furthermore, some collective communication operations like broadcast and parallel prefix are also provided. To become more flexible, PUB allows to create independent BSP objects each representing a virtual BSP computer. Goudreau et al.[GHL+96] defined an interface for BSP libraries call "BSP Worldwide Standard" which is implemented in the Oxford BSP toolset[HDM97]. You can also use the BSP Worldwide Standard for programming with PUB (refer to Section 1.5). A performance comparison with other communication libraries and some implementation details can be found in [BJvR98].

The PUB–Library is available for several platforms

- Parsytec Gigacluster - GCel (Parix)
- Parsytec GC/PowerPlus (Parix)
- Parsytec CC (Embedded Parix)
- IBM SP2 (MPL or MPI)
- NOW (Network of Workstations) Solaris (TCP/IP)
- NOW Linux (TCP/IP)

Furthermore, PUB can be easily adapted to all platforms which offer an MPI implementation.

PUB also contains a BSP simulator to run parallel programs for easy debugging on simple workstations (or PCs) with

- Sun Solaris
- Windows 32
- IBM OS/2

1.2 Environment

Installation

For each available platform there exists one gzipped tar-archive at `http://www.uni-paderborn.de/~pub`. Please load this file and untar it. This archive contains the following files:

- `pub.h`: PUB include file
- `bspww.h`: include file, if you want to use the BSP worldwide standard
- `libpub.a`: PUB library file which you have to link to your program

- libpub_d.a: PUB–library containing debug information
- several *.c example files

Store these files in one directory which we call <pubdir> in the following.

Compiling and Linking

If you want to compile a program which uses PUB functions you have to include the file pub.h in your C–file (or file bspww.h if you want to use BSP Worldwide functions, see 1.5). Furthermore, you have to comile and link your program with the command

```
<cc> <prg>.c -I<pubdir> -L<pubdir> -lpub <linklibs> -o <prg>
```

Here, <cc> is the C-compiler you want to use, <prg> is your program file and <linklibs> are the system libraries which have to be linked to your program depending on the platform. The following table shows which compiler may be used and which libraries has to be linked to your program.

If you want to link the debug version of PUB which contains more assertions but which is also slower you have to call

```
<cc> <prg>.c -I<pubdir> -L<pubdir> -lpub_d <linklibs> -o <prg>
```

Platform	C–compiler	Libraries	Start command
Parsytec GCel	ancc.px	-	ccsalloc or ccsrun
Parsytec GC/PP	ancc.px	-	ccsalloc or ccsrun
Parsytec CC	ancc.px	-	ccsalloc or ccsrun
Intel Paragon	???	-	???
IBM SP2	mpcc	-	poe or job_submit
NOW Sun Solaris	cc or gcc	pthread posix4 m nsl socket	pubrun
Simulator Sun Solaris	cc or gcc	pthread posix4 m nsl socket	
Simulator Windows 32	Visual C++	-	program name
Simulator IBM OS2	IBM VAC++ or gcc (emx)	-	program name

Networks of Workstations

PUB can be used on networks of workstations running Solaris or Linux. You have to compile these programs as presented in the previous section. For running these programs on your network you have to create a file machines in your current working directory which consists of the computing nodes on which your program will be executed. Example for the file machines:

```
atm1
atm2
fast1
fast2
```

Thus, this file is similar to the *machines* file of MPI. One difference to MPI is that the first machine, your program is started on, is the one in the first line of the **machines** file. Using MPI the first copy of your program is running on the machine you are currently working on.

Now you can start your program with **pubrun [-procs:n] <program-name>**.

PUB in Paderborn

The include and example files are installed at the University of Paderborn in the dirctory ~/pub/include resp. ~pub/examples. The library files can be found in ~pub/lib/<arch>, where <arch> is one of GCEL,GCPP,CC, SOLARIS,LINUX,SUNSIM.

1.3 Programming with PUB

Initialization

The following program shows how PUB is initialized.

```
1  #include <pub.h>
2  int main (int argc, char* argv[]) {
3     tbsp bsp;
4     int pid, nprocs;
5     bsplib_init (BSPLIB_STDPARAMS, &bsp);
6     nprocs = bsp_nprocs (&bsp);
7     pid = bsp_pid (&bsp);
8     printf ("Hello world from processor %d of %d!\n", pid, nprocs);
9     bsplib_done ();
10    return 0;
11 }
```

Before any function of the PUB–Library can be used, you have to call **bsplib_init** (line 5). The struct **bsp** is initialized here. In the following, the first parameter of each PUB function is **&bsp**. In line 6 and 7 we get some information about the machine: **bsp_nprocs** returns the number of processors and **bsp_pid** returns the own processor id. The processor id is in the range $0, \ldots, nprocs - 1$. In line 9 we stop PUB by a call to **bsplib_done**. In the follwing program examples we always use **pid** as processor id and **nprocs** as number of available processors.

Message passing

According to the BSP model all messages are received during the synchronization and cannot be read before.

In the following program segment each processor sends a message to "its right neighbour" (that means to processor **pid+1**) and to its left neighbour.

```
int i, destination, data;
data = 1;
destination = (pid+1) % nprocs;
bsp_send (&bsp, destination, &data, sizeof (int));
data = 2;
destination = (pid-1) % nprocs;
bsp_send (&bsp, destination, &data, sizeof (int));
bsp_sync (&bsp);
```

The function **bsp_send** has the parameters **&bsp**, the destination processor **destination**, a pointer

`&data` to the data which will be sent, and the size of the data (`sizeof (int)`). This send operation is a buffered send. This means, after `bsp_send` has been called (line 4) the data can be reused by the sender (line 5) - the receiver will get the value 1. Despite of `bsp_send` there are the operations `bsp_sendmsg` and `bsp_hpsend` which can be used in special situtations if more performance is needed. See the function reference for more information.

After calling `bsp_sync` all messages sent in the last superstep can be read with `bsp_getmsg` and `bsp_findmsg`. The operation `bsp_nmsgs` gives the number of received messages in the last superstep. The following code will iterate all received message:

```
t_bspmsg *msg;
int i, *data;
for (i=0; i<bsp_nmsgs (&bsp); i++) {
   msg = bsp_getmsg (&bsp, i);
   data = (int*) bspmsg_data (msg);
   // evaluate data
}
```

To iterate all messages from the processor `sender`, call `bsp_findmsg` until it returns NULL:

```
t_bspmsg *msg;
int i=0;
while (msg=bsp_findmsg (&bsp, sender, i)) {
   // use msg
   i++;
}
```

To work on a message you can use the operations `bspmsg_data`, `bspmsg_size` and `bspmsg_src`. The messages of the last supersteps are available until the next synchronization call. At this point the memory used for these messages will be deallocated. So, if you want to use these messages later you have to copy them.

Remote memory access

Another way of communication is remote memory access: After every processor has registered a variable for direct access, all processors can read and write the value of the variable on other processors.

```
1 int array[100], privateArray[100];
2 src = 0;
3 dest = 0;
4 offset = 0;
5 bsp_push_reg (&bsp, array, sizeof(array));
6 bsp_get (&bsp, src, array, offset, privateArray, sizeof(array));
7 bsp_put (&bsp, dest, &privateArray[1], array, bsp_pid(bsp)*sizeof(int),
  sizeof(int));
8 bsp_sync (&bsp);
9 bsp_pop_reg (&bsp, array);
```

In this example every processor gets a copy of `array` on processor 0 in his own `privateArray` (`bsp_get`). After that each processor writes `privateArray[1]` into `array[pid]` on processor 0 (`bsp_put`, here bsp_pid(bsp)*sizeof(int) is an offset for array). All get and put operations are executed during the synchronization and all get operations are served before a put overwrites a value. Therefore, you could exchange lines 6 and 7 in the above example and you would get the same result.

The operations `bsp_hpget` and `bsp_hpput` can be used if more performance is needed. But there are several restrictions for these "high performance" operations. See the function reference for more information.

The size of a registered variable can be different on each node, but each processor of the group has to call bsp_push_reg (use size 0 if the node has no data). If several regions are registered for remote access all processors have to register them in the same order.

Types of Synchronization

PUB contains two different types of synchronization:

`bsp_sync (bsp)`: standard synchronization
`bsp_oblsync (bsp, nmsgs)`: oblivious synchronization.

The oblivious synchronization should be used if the programmer knows the number of messages each processor will receive in a superstep. E.g., if each processor sends a message to its right neighbour then every processor knows that it will receive exactly one message. Thus, in the oblivious synchronization each processor waits until `nmsgs` are received. This type of synchronization is much faster than the other one since no additional communication is needed. The standard synchronization should be used if the number of messages to receive is unknown. Supersteps with standard synchronization can alternate with oblivious synchronizations, but within one superstep each processor has to use the same type of synchronization.

What is the *number of messages* (`nmsgs`) a processor waits in `bsp_oblsync` for?

1. Each message which will be found in the message queue. (Sent with `bsp_send`, `bsp_msgsend`, `bsp_hpsend`, or according broadcast operations (`bsp_g...`))
2. Each `bsp_put` (or `bsp_hpput`) produces "one message" at the destination. Each `bsp_get` (or `bsp_hpget`) produces "one message" at source and destination.
3. Each prefix or sum operation produces "one message" at each processsor involved. This also holds for `bsp_reduce` where only one processor gets a result.

Broadcasts

In order to send a message to more than one processor the PUB-Library provides a number of broadcast functions.

Again the messages are received during the synchronization and cannot be read before.

The following program fragment shows a simple use of `bsp_gsend`, wherein each processor sends a message to each other including itself.

```
1 char message[MAXLENGTH];
2 sprintf(message, "Hello from processor %d", bsp_pid(&bsp));
3 bsp_gsend(&bsp, 0, bsp_nprocs(&bsp)-1, message, sizeof(message));
4 bsp_sync(&bsp);
```

The two arguments `0` and `bsp_nprocs(&bsp)` of `bsp_gsend` in line 3 define the "processor group" to send the message to. The first of them is the lowest and the latter the highest processor id of the so defined interval. Therefore line 3 sends to all processors `sizeof(message)` bytes read from `message`.

There is an unbuffered version of `bsp_gsend`, namely `bsp_hpgsend`. Using this "high performance broadcast" you must not reuse the variable `message` until the next synchronization.

Furthermore there is the function family `bsp_lsend`, which allows to send a message to a given list of processors.

Prefix and Sums

The PUB-Library offers a set of functions to perform “global operations”. A “global operation” can be applied to data stored by a subset of nodes.

```
1  tbsp bsp;
2  int pid, nprocs;
3  int data;
4  int reduce, allreduce, scan, allscan1, allscan2;
5  reduce = allreduce = scan = allscan1 = allscan2 = 0;
6
7  bsplib_init( BSPLIB_STDPARAMS, &bsp );
8  pid = bsp_pid( &bsp ); nprocs = bsp_nprocs( &bsp );
9  data = pid;
10 bsp_reduce( &bsp, 0, 0, nprocs-1, &data, &reduce, 1, &bspop_addint, NULL );
11 bsp_scan( &bsp, 1, 0, nprocs-1, &data, &scan, 1, &bspop_addint, NULL );
12 bsp_allreduce( &bsp, 2, 0, nprocs-1, &data, &allreduce, 1, &bspop_addint,
   NULL );
13 bsp_allscan( &bsp, 3, 0, nprocs-1, &data, &allscan1, &allscan2, 1,
   &bspop_addint, NULL );
14 bsp_sync( &bsp );
15
16 bsplib_done();
```

The variable `data` defined in line 3 is the node's operand to the global operation. In line 4 and 5 the variables for the results are defined and initialized. Line 9 sets the data for each node to its processor id. Each of the next lines starts a prefix or sum operation.

In this example the predefined operation `bspop_addint` is used, which provides addition of integers. Also predefined are operation to take the maximum and minimum of both integers and doubles (self-evident there's bspop_adddouble).

The functions `bsp_reduce`, `bsp_scan` and `bsp_allreduce` take as parameters:

1. a pointer to a bsp-object
2. a tag which has to be equal on all nodes but unique in this superstep
3. the lowest node id involved
4. the highest node id involved (forming the interval with 3.)
5. address of the data vector
6. pointer to a buffer wherein the result is to be stored
7. length of the data vector in elements (not bytes!)
8. the operation
9. user defined pointer which will be passed to the operation function

Function `bsp_allscan` differs slightly by taking two result buffers (here `allscan1` and `allscan2`).

`bsp_reduce`: determines the total sum of the data vectors of all processors. Only the processor with the highest id stores the result after synchronization. Nevertheless all other processors within the group has to provide a correct pointer to resultbuf. So in this example processor `nprocs-1` stores the sum of all pid's (`reduce`=0+1+..+nprocs-1) by reaching line 14.

`bsp_scan`: determines the prefix sum of the data vectors. Reaching line 14 each processor holds accordingly the sum of its and all lower pid's (`scan`=0+1+..+pid).

`bsp_allreduce`: determines the sum of the vectors of all (involved) processors and redistributes this sum to all (involved) processors. Each processor in our example holds the sum of all pid's in `allreduce`.

`bsp_allscan`: determines the prefix sum and the total sum of the data vectors. After the synchronization the first result buffer holds the prefix sum (like `bsp_scan`) and the second result buffer holds the total sum. That is `allscan1`=0+1..+pid and `allscan2`=0+1..+nprocs-1 in the given example.

To each of these functions there is a 'l'-version (`bsp_lreduce` and so on) taking a list of nodes instead of the lowest and highest involved node id.

Attention: The result is calculated asynchronously, so neither modify the `data` nor use the result buffers before synchronization!

How to define a global operation?

Let's see how `bspop_addint` is defined:

```
tbspop bspop_addint = {Xbspop_Xaddint, sizeof(int)};
```

The structure `tbspop` has two members. The first is a pointer to a function of type `pbspop_function` taking four arguments as we'll see. The second member specifies the size of the used operands in bytes.

```
void Xbspop_Xaddint( void* A, void* BAndResult, int count, void* Parameter )
{
   int i;
   for (i=0;i<count;i++)
     ((int*)(BAndResult)) [i] += ((int*)(A)) [i];
}
```

A `pbspop_function` takes four arguments, wherein the first two are void-pointers to the operand-vectors `A` and `B`. To B the operations result is stored. The parameter `count` gives the number of elements per operand-vectors. The last argument is a user defined pointer which is taken as the last argument of one of the prefix functions (`bsp_scan`, ...) and passed to the operation functions without any changes.

Attention: Defined operations have to be commutative!

Subgroups

In many parallel algorithms there are subproblems that can be solved on a subset of processors. Although the BSP model is based on global synchronization, PUB offers the opportunity to partition the set of processors into pairwise disjoint subsets. Each subset acts like an independent BSP computer.

```
tbsp subbsp;
int partition[2];
partition[0] = bsp_nprocs(bsp) / 2;
partition[1] = bsp_nprocs(bsp);
bsp_partition (bsp, &subbsp, 2, partition);
// do calculations, use &subbsp as a pointer to the bsp-object
bsp_done (&subbsp);
```

First, we create an integer–array describing a partition of the nodes. The entries of the array are the lowest processor ids of the nodes of each following subset. In the example we create the subsets [0:nprocs/2-1], [nprocs/2:nprocs(bsp)-1]. To get a correct partition, the array has to be sorted and the last entry must be equal to `bsp_nprocs(bsp)`. In line 5 we create the subgroups and get a new BSP object `subbsp`. We can now use this BSP object independently in all two subsets for all PUB operations (except for `bsplib_done`). To break up the subgroups we call `bsp_done`. This waits until all subgroups have finished their work and invalidates the subbsp object.

It is allowed to send messages to a parent BSP object, but you must not call `bsp_sync` using the parent object.

The creation of subgroups can be nested, but they must be created and destroyed in last-in first-out order (like a stack), so you can only divide the newest subgroup and you have to delete the newest subgroup before any older one.

You can also create one "subgroup" of the same size as the actual BSP object by `bsp_dup` to call a subprogram which does not interfere with your superstep:

```
1  void subAlg() {
2     tbsp subbsp;
3     bsp_dup (bsp, & subbsp);
4     // do work, send messages, ...
5     bsp_sync (& subbsp);
6     // do work, send messages, ...
7     bsp_done (& subbsp);
8  }
9
10    bsp_send (...)
11    subAlg();
12    bsp_sync (bsp);
```

If you do not use a subgroup in the subalgorithm, your message sent in the main program (line 10) will be received in the subalgorithm.

If you want to use different BSP objects within different threads you have to use `bsp_threadsave_dup` and `bsp_threadsave_done`, see also the following section.

Threads

The PUB–Library supports the use of threads if the underlying operating system supports threads and message passing simultaneously (all except IBM SP/2). There is a subset of POSIX 1003.1c threads and counting semaphores which we have implemented on the systems which do not offer this standard. You should use these thread functions instead of other operating system thread libraries to create architecture independent programs. The simulator for debugging does not support threads created by functions other than the simulators pthread_create.

The following functions are supported:

Threads: pthread_attr_init, pthread_attr_setstacksize, pthread_create

Mutex semaphores: pthread_muttex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock, pthread_mutex_destroy

Counting semaphores: sem_init, sem_post, sem_wait, sem_trywait, sem_destroy

All functions of the PUB–Library are thread–safe, but one BSP object and all its subgroups must not be used by more than one thread. If you want to use BSP objects in different threads, create them with `bsp_threadsave_dup`.

1.4 Simulator

The PUB–Simulator can be used for developing and debugging of parallel programs. Here you have the advantage that you can use your usual development tools on your workstation or PC. Currently the operating systems MS Windows 32, IBM OS/2 and Sun Solaris are supported. The different processors of a parallel machine are simulated by different threads on the one–processor machine. If you want to use this simulator you have to do only one thing: you have to register all global variables in your program like in the following example.

```
int globalVar;

int main (int argc, char* argv[]) {
   t_bsp bsp;
   bsplib_init (BSPLIB_STDPARAMS, &bsp);
   bsp_globalreg (&bsp, &globalVar, sizeof (globalVar));
   // main program
   bsplib_done ();
   return 0;
}
```

If you compile this program for a real parallel machine you do not need to remove the registrations since they are automatically removed by empty-operations if not needed.

1.5 BSP Worldwide Standard

If you want to use the BSP Worldwide Standard (see [GHL+96] for a function reference) instead of the operations offered by PUB you have to include the file `bspww.h` instead of `pub.h` in your program files. Compiling and linking can be done as described in 1.2.

Attention: In opposition to PUB, BSP Worldwide allows calls to bsp_nprocs before the initialisation, e.g. a call like bsp_begin(bsp_nprocs()). This also works in PUB, but bsp_nprocs() returns BSPLIB_ALLPROCS instead of the real number of processors, so you cannot do computation with the result.

2 Reference

2.1 List of functions

Initialization, Synchronization and Enquiries

name	page	description
bsplib_init	14	initialize the PUB–Library
bsplib_done	14	exit PUB–Library, frees resources
bsplib_params_init	14	initializes the PUB–Library parameters
bsp_nprocs	15	get number of processors
bsp_pid	15	get own processor–id
bsp_time	15	get time (double)
bsp_sync	15	synchronization
bsp_oblsync	15	synchronization with known number of messages
bsp_abort	16	abort the program on all processors

Message Passing

bsp_send	17	send a memory block
bsp_nmsgs	17	get number of received messages
bsp_getmsg	17	get pointer to received message
bsp_findmsg	17	find received message according to sender
bspmsg_data	18	get pointer to data of a message
bspmsg_size	18	get size of a message
bspmsg_src	18	get id of the sender–processor

Remote Memory Access

bsp_push_reg	19	register a variable for remote access
bsp_pop_reg	19	delete registration of a variable
bsp_put	19	remote write to a variable of another processor
bsp_get	20	remote read from a variable of another processor

Subgroups and BSP objects

bsp_dup	21	duplicate bsp object
bsp_partition	21	create subgroups
bsp_done	21	end subgroup or double bsp object
bsp_threadsafe_dup	22	duplicate bsp object for a new thread
bsp_threadsafe_done	22	remove duplicated bsp object

Broadcast

bsp_gsend	23	send a memory block to an interval of nodes
bsp_lsend	23	send a memory block to a list of nodes
bsp_gsendmsg	23	send a message to an interval of nodes
bsp_lsendmsg	23	send a message to a list of nodes
bsp_hpgsend	23	unbuffered send of a memory block to an interval of nodes
bsp_hplsend	23	unbuffered send of a memory block to a list of nodes

Prefix and Sum

bsp_reduce	24	determine the sum over an interval of processors
bsp_scan	24	determine the prefix sum over an interval of processors
bsp_allreduce	24	determine the sum over an interval of processors and broadcast this value
bsp_allscan	24	determine prefix sum and broadcast the sum to the interval processors
bsp_lreduce	25	determine the sum over arbitrary processors
bsp_lscan	25	determine the prefix sum over arbitrary processors
bsp_lallreduce	25	determine the sum over arbitrary processors and broadcast this value
bsp_lallscan	25	determine prefix sum and broadcast the sum to arbitrary processors

High Performance

bsp_hpget	20	remote unbuffered read from a variable of another processor
bsp_hpput	19	remote unbuffered write to a variable of another processor
bsp_createmsg	27	create a new message
bsp_sendmsg	17	send a message
bsp_hpsend	17	unbuffered send of memory blocks

Information

bsp_parameter	28	get all bsp*–parameter
bsp_b	28	get bsp*–parameter B
bsp_g	28	get bsp–parameter g
bsp_l	28	get bsp–parameter L
bsp_s	28	get bsp–parameter s (time per instruction)

Simulator

bsp_globalreg	29	register global variables

2.2 Description of functions

Initialization, Synchronization and Enquiries

bsplib_init

void bsplib_init (t_bsplib_params* parameter, t_bsp* bsp)

parameter	type	description
parameter	t_bsplib_params*	parameter
bsp	t_bsp*	pointer to the bsp–object to initialize

This function initializes the PUB–Library. It must be called before any other function of the library (except bsplib_param_init) can be used.

Parameter can be BSPLIB_STDPARAMS or a pointer to a struct t_bsplib_params initialized with bsplib_params_init.

bsplib_done

void bsplib_done ()

This function cleans up all PUB resources. After the call of bsplib_done you must not use any function of the PUB–Library.

Attention: In the Unix–cluster–implementation a call to bsplib_done terminates the program. Therefore, no statement after bsplib_done will be executed.

bsplib_params_init

void bsplib_params_init (t_bsplib_params* parameter)

parameter	type	description
parameter	t_bsplib_params*	parameter

This function initializes the parameter structure. After this you can modify some of the values given in the following table. Then you can use this parameter pointer to initialize the PUB–Library using `bsplib_init` accordingly.

struct t_bsplib_params

member	type	description
info	BOOL	operate in info–mode
profile	YES, NO, IFNOTINFILE	calculate bsp parameters
monitor	BOOL	monitor bsp work
procs	int	number ob processors to use or BSPLIB_STDPARAMS
buffersize	int	size of buffer
buffering_on	BOOL	TRUE if buffering on
multiRecvThreads	BOOL	TRUE if one receive thread for every other node will be started

bsp_nprocs

int bsp_nprocs (t_bsp* bsp)

parameter	type	description
bsp	t_bsp	pointer to BSP object

This function returns the number of nodes in the group defined by the BSP object.

bsp_pid

int bsp_pid (t_bsp* bsp)

parameter	type	description
bsp	t_bsp	pointer to BSP object

This function returns the id of the node in the range 0,...,`bsp_nprocs`-1.

bsp_time

double bsp_time (t_bsp* bsp)

parameter	type	description
bsp	t_bsp	pointer to BSP object

This function returns the time since bsplib_init was called in seconds. Althought this time is reseted on all nodes in bsplib_init, the clocks of the nodes may run asynchronously, so times of different nodes cannot be compared.

bsp_sync

void bsp_sync (t_bsp* bsp)

parameter	type	description
bsp	t_bsp	pointer to BSP object

This function synchronizes the BSP group. It blocks the current thread until:

- all other nodes in the actual group has called bsp_sync
- all messages sent to this node in the current superstep has been received
- all prefix–operations has been finished

bsp_oblsync

void bsp_oblsync (t_bsp* bsp, int nr_msgs)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
nr_msgs	int	number of messages received in the last superstep

This function synchronizes the BSP group. It blocks the current thread until `nr_msgs` messages has been received. This function is faster than `bsp_sync` because no barrier synchronization is done, but the user must know exactly how many messages every node will receive. What is the *number of messages* (`nmsgs`) a processor waits in `bsp_oblsync` for?

1. Each message which will be found in the message queue. (Sent with `bsp_send`, `bsp_msgsend`, `bsp_hpsend`, or according broadcast operations (`bsp_g...`))
2. Each `bsp_put` (or `bsp_hpput`) produces "one message" at the destination. Each `bsp_get` (or `bsp_hpget`) produces "one message" at source and destination.
3. Each prefix or sum operation produces "one message" at each processsor involved. This also holds for `bsp_reduce` where only one processor gets a result.

Attention: nr_msgs must be equal to the real number of received messages or the behaviour is undefined. Every node must use the same type of synchronization, do not mix bsp_sync and bsp_oblsync in the same superstep!

bsp_abort

void bsp_abort (t_bsp* bsp, char* format, ...)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
format	char*	format of the output (like printf)
...		additional arguments

This function prints an error message to stderr and stops the program on all nodes.

Message Passing

bsp_send, bsp_sendmsg, bsp_hpsend

void bsp_send (t_bsp* bsp, int dest, void* buffer, int size)
void bsp_sendmsg (t_bsp* bsp, int dest, t_bspmsg* msg, int size)
void bsp_hpsend (t_bsp* bsp, int dest, void* buffer, int size)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
dest	int	id of the destination node
buffer	void*	pointer to a memory address to send
msg	t_bspmsg*	pointer to the message which will be sent
size	int	size of the block or message

These operations send a single message to processor `dest`. The `dest` processor will receive this message after the next synchronization and can access this message with `bsp_getmsg` or `bsp_findmsg`.

`bsp_send` sends the memory contents of address `buffer` with size `size`. After calling this function the `buffer` can be reused by the user and may be overwritten or freed (buffered send).

`bsp_sendmsg` sends a message of type t_bspmsg*. Such a message has to be created by `bsp_createmsg`. After calling this operation the user must not access `msg` any more; deallocating memory for this message is done by PUB.

`bsp_hpsend` sends like `bsp_send` the memory contents of `buffer`. Until the next synchronization `buffer` must not be overwritten by the user since this is an unbuffered send.

Which send operation should be used in which situation?

If there is not much communication in your program or superstep	`bsp_send`
If there is a lot of data not in continous memory for the same destination processor	`bsp_sendmsg`
If you are sure that you do not want to change the message until the next superstep and your message is already in continous memory	`bsp_hpsend`

bsp_nmsgs

int bsp_nmsgs (t_bsp* bsp)

parameter	type	description
bsp	t_bsp	pointer to BSP object

This function returns the number of messages received in the last superstep (before the last call of bsp_sync). These messages can be accessed with the operations `bsp_findmsg` or `bsp_getmsg`.

bsp_findmsg, bsp_getmsg

t_bspmsg* bsp_findmsg (t_bsp* bsp, int proc_id, int index)
t_bspmsg* bsp_getmsg (t_bsp* bsp, int index)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
proc_id	t_proc_id	id of the source–node of the message
index	int	index of the message, beginning from 0

These functions return a pointer to a message received in the last superstep (before the last call of bsp_sync). Such a message was sent by `bsp_send`, `bsp_sendmsg`, `bsp_hpsend` or by one of the broadcast operations `bsp_gsend`, `bsp_gsendmsg`, `bsp_hpgsend`, `bsp_lsend`, `bsp_lsendmsg`, `bsp_hplsend`. All messages in the actual message list are valid until the next synchronization. If you want to use a message later you have to copy it. For working on messages the operations `bspmsg_data`, `bspmsg_size` or `bspmsg_src` can be used.

`bsp_get`: all messages in the message list are numbered from 0 to `bsp_nmsgs`-1. Using `bsp_get` you get message at position `index`.

`bsp_find`: with `bsp_find` you can search in your message list for messages which came from processor `proc_id`. If these are n then they are numbered from 0 to $n-1$. If `index` $> n-1$ `bsp_find` returns NULL.

bspmsg_data

void* bspmsg_data (t_bspmsg* msg)

parameter	type	description
msg	t_bspmsg*	pointer to the message

This function returns a pointer to the data of a message.

bspmsg_size

int bspmsg_size (t_bspmsg* msg)

parameter	type	description
msg	t_bspmsg*	pointer to the message

This functions returns the number of bytes the message contain.

bspmsg_src

int bspmsg_src (t_bspmsg* msg)

This function returns the node–id of the sender of the message or -1 if the message was not received. (This means if the message was created using `bsp_createmsg` by the processor itself.)

parameter	type	description
msg	t_bspmsg*	pointer to the message

Remote Memory Access

bsp_pop_reg

void bsp_pop_reg (t_bsp* bsp, void* ident)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
ident	void*	local address of the variable, same as in call bsp_push_reg

This function deletes a variable registered with bsp_push_reg for global access.

Attention: All nodes must call this function in the same order.

bsp_push_reg

void bsp_push_reg (t_bsp* bsp, void* ident, int size)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
ident	void*	local address of the variable, also identifier for bsp_get and bsp_put
size	int	size of variable, can be different on different nodes

This function registers a variable for global access. All nodes must call bsp_push_reg with `ident` pointing to a local variable. The size of the variabale (`size` bytes) can be different on each node.

Then every node can access the variables on other nodes with `bsp_get` (`bsp_hpget`) and `bsp_put` (`bsp_hpput`) using the local address as an identifier.

Attention: If different variables have to be registered all processors have to call the `bsp_push_reg` functions in the same order.

bsp_put, bsp_hpput

void bsp_put (t_bsp* bsp, int destPID, void* src, void* dest, int offset, int nbytes)

void bsp_hpput (t_bsp* bsp, int destPID, void* src, void* dest, int offset, int nbytes)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
destPID	int	id of destination node
src	void*	pointer to source memory
dest	void*	pointer to destination variable
offset	int	offset in destination variable
nbytes	int	number of bytes to copy

These functions copy `nbytes` bytes from local memory `src` to variable `dest` (with offset `offset`) on node `destPID`.

`bsp_put`: The destination is written on in the synchronization after all bsp_get operations are finished.

`bsp_hpput`: This is the high performance version of bsp_put. The copy is done asynchronously and unbuffered, so it is finished after the next call to bsp_sync (or bsp_oblsync), the buffer src must not be changed before. Do not use this function if destination–buffer is used in this super–step because the time the destination is written is undefined.

bsp_get, bsp_hpget

void bsp_get (t_bsp* bsp, int srcPID, void* src, int offset, void* dest, int nbytes)

void bsp_hpget (t_bsp* bsp, int srcPID, void* src, int offset, void* dest, int nbytes)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
srcPID	int	id of source node
src	void*	pointer to source–variable
offset	int	offset in source–variable
dest	void*	pointer to destination memory
nbytes	int	number of bytes to copy

This function copies `nbytes` bytes from the variable `src` (with offset `offset`) on processor `srcPID` to the local memory address `dest`.

`bsp_get`: The source is copied when node srcPID calls `bsp_sync` (or `bsp_oblsync`) before any `bsp_put` is executed.

`bsp_hpget`: This is an unbuffered remote memory access on source and on destination. That means both `src` and `dest` must not be overwritten until the next synchronization.

Attention: The copy is done asynchronously, so it is finished after the next call to bsp_sync (or bsp_oblsync), the buffer dest must not be used before.

Subgroups and BSP objects

bsp_dup

void bsp_dup (t_bsp* bsp, t_bsp* dup)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
dup	t_bsp*	pointer to the memory for the new BSP object

This function initializes a new BSP object as a copy of bsp. The new BSP object is a subgroup of bsp containing all nodes, so

```
int partition=bsp_nprocs( bsp );
bsp_partition( bsp, & dup, 1, & partition );
```

is the same. To free the new object use bsp_done. See bsp_partition for details.

Attention: The new bsp object dup is not independent, it belongs to bsp, so you must not use bsp and dup in different threads, use bsp_threadsafe_dup to get a thread–save copy

bsp_partition

void bsp_partition (t_bsp* bsp, t_bsp* sub, int nr, int* partition)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
sub	t_bsp*	pointer to memory for the new subgroup BSP object
nr	int	number of entries in partition
partition	int*	pointer to node–ids defining the subgroups, the new groups contain node–ids [0, partition[0]-1], . . . , [partition[nr-2], partition[nr-1]-1], partition[nr-1] must be the number of nodes in the group bsp

This function creates a new subgroup of bsp and initializes a new BSP object *sub. The new BSP object can be used independently to all bsp–functions. For example a sub–algorithm can create a subgroup containing all nodes of the BSP object to avoid conflicts with other algorithm.

Although the old BSP object bsp can still be used, you must not create other subgroups of bsp until sub is destroyed with bsp_done. Using of subgroups is like a stack, only the lowest subgroup is allowed to generate other subgroups.

bsp_done

void bsp_done (t_bsp* bsp)

parameter	type	description
bsp	t_bsp	pointer to BSP object

This function destroys a subgroup created with bsp_dup or bsp_partition and frees the BSP object.

You must not send a message to an already destroyed BSP object. The behavior of the PUB–Library will be undefined.

bsp_threadsafe_dup

void bsp_threadsafe_dup (t_bsp* bsp, t_bsp* dup)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
dup	t_bsp*	pointer to the memory for the new BSP object

This function initializes a new BSP object as a copy of bsp. The new BSP object is independent from the old BSP object, so it can be used in a different thread.

To free the new object use bsp_threadsafe_done.

bsp_threadsafe_done

void bsp_threadsafe_done (t_bsp* bsp)

parameter	type	description
bsp	t_bsp	pointer to BSP object

This function destroys a subgroup created with bsp_threadsafe_dup and frees the BSP object.

You must not send a message to an already destroyed BSP object. The behavior of the PUB–Library will be undefined.

Broadcast

bsp_gsend, bsp_gsendmsg, bsp_hpgsend, bsp_lsend, bsp_lsendmsg, bsp_hplsend

void bsp_gsend (t_bsp* bsp, int proc_lo, t_proc_id proc_hi, void* buffer, int size)

void bsp_gsendmsg (t_bsp* bsp, int proc_lo, t_proc_id proc_hi, t_bspmsg* msg, int size)

void bsp_hpgsend (t_bsp* bsp, int proc_lo, t_proc_id proc_hi, void* buffer, int size)

void bsp_lsend (t_bsp* bsp, int* dest_procs, int ndest, void* buffer, int size)

void bsp_lsendmsg (t_bsp* bsp, int* dest_procs, int ndest, t_bspmsg* msg, int size)

void bsp_hplsend (t_bsp* bsp, int* dest_procs, int ndest, void* buffer, int size)

parameter	**type**	**description**
bsp	t_bsp*	pointer to the BSP object
proc_lo	int	lowest id of the desination interval
proc_hi	int	highest id of the destination interval
dest_procs	int*	array of destination processors
ndest	int	number of ids in dest_procs
buffer	void*	pointer to the block to send
msg	t_bspmsg*	pointer to the message
size	int	size of the memory block or message

These functions send a message to a group of processors. The group of receiving processors is specified by the interval [`proc_lo`:`proc_hi`] or a list of processor ids. The message will be available at the receivers after a synchronization (`bsp_sync` or `bsp_oblsync`).

`bsp_gsend` and `bsp_lsend` send the memory block pointed by `buffer` which has the size `size`. After calling this functions `buffer` can be reused by the user immediately (see also `bsp_send`).

`bsp_gsendmsg` and `bsp_lsendmsg` send a message `msg` of type t_bspmsg* which was created by `bsp_createmsg`. After calling one of this functions `msg` is not available for the user anymore (see also `bsp_sendmsg`).

`bsp_hpgsend` and `bsp_hplsend` send a memory block like `bsp_gsend` and `bsp_lsend`, but after this call `buffer` must not be reused by the user until the next synchronization (see also `bsp_hpgsend`, `bsp_hplsend`).

Prefix and Sum

bsp_reduce, bsp_scan, bsp_allreduce, bsp_allscan

void bsp_reduce (t_bsp* bsp, int tag, int group_lo, int group_hi, void* databuf, void* resultbuf, int count, pbspop assop, void* paramter)

void bsp_scan (t_bsp* bsp, int tag, int group_lo, int group_hi, void* databuf, void* resultbuf, int count, pbspop assop, void* parameter)

void bsp_allreduce (t_bsp* bsp, int tag, int group_lo, int group_hi, void* databuf, void* resultbuf, int count, pbspop assop, void* paramter)

void bsp_allscan (t_bsp* bsp, int tag, int group_lo, int group_hi, void* databuf, void* result1buf, void* result2buf, int count, pbspop assop, void* paramter)

parameter	type	description
bsp	t_bsp*	pointer to the bsp_object
tag	int	tag to distinguish several operations from each other, must be equal on all nodes and unique in this superstep
group_lo	int	id of lowest involved node
group_hi	int	id of highest involved node
databuf	void*	pointer to the data vector
resultbuf	void*	pointer to the buffer for the result (prefix or sum)
result1buf	void*	pointer to the buffer for the prefix result
result2buf	void*	pointer to the buffer for the total result
count	int	length of the vector in elements
assop	pbspop	kind of operation
parameter	void*	userdefined parameter, not used by PUB–Library

These functions calculate the result of a global operation (e.g. sum). Each node from `group_lo` to `group_hi` specifies a vector of `count` data elements (`count` must be the same on all nodes) starting at address `databuf`. After the next synchronization the the result is in buffer `resultbuf` or in `result1buf` and `result2buf`.

The PUB–Library offers the following basic operations: bspop_adddouble, bspop_maxdouble, bspop_mindouble, bspop_minmaxdouble for elements of type double and bspop_addint, bspop_maxint, bspop_minint, bspop_minmaxint for integers.

Attention: The result is calculated asynchronously, so neither modify `databuf` nor use `resultbuf` before synchronization!

`bsp_reduce`: determines the total sum of the vectors of all processors. Only processor `group_hi` stores the result after synchronization. Nevertheless all other processors within the group has to provide a correct pointer to `resultbuf`.

`bsp_scan`: determines the prefix sum of the vectors.

`bsp_allreduce`: determines the sum of the vectors of all processors and redistributes this sum to all processors.

`bsp_allscan`: determines the prefix sum and the total sum of the vectors. After the synchronization `result1buf` holds the prefix sum (like `bsp_scan`) and `result2buf` holds the total sum.

bsp_lreduce, bsp_lscan, bsp_lallreduce, bsp_lallscan

void bsp_lreduce (t_bsp* bsp, int tag, int* dstlist, int ndst, void* databuf, void* resultbuf, int count, pbspop assop, void* paramter)

void bsp_lscan (t_bsp* bsp, int tag, int* dstlist, int ndst, void* databuf, void* resultbuf, int count, pbspop assop, void* paramter)

void bsp_lallreduce (t_bsp* bsp, int tag, int* dstlist, int ndst, void* databuf, void* resultbuf, int count, pbspop assop, void* paramter)

void bsp_lallscan (t_bsp* bsp, int tag, int* dstlist, int ndst, void* databuf, void* result1buf, void* result2buf, int count, pbspop assop, void* paramter)

parameter	**type**	**description**
bsp	t_bsp*	pointer to the bsp_object
tag	int	tag to distinguish several operations from each other, must be equal on all nodes and unique in this superstep
dstlist	int*	array of involved node ids, pairwise different
ndst	int	number of elements in dstlist
databuf	void*	pointer to the data vector
resultbuf	void*	pointer to the result buffer
result1buf	void*	pointer to the buffer for the prefix result
result2buf	void*	pointer to the buffer for the total result
count	int	length of the vector in elements
assop	pbspop	kind of operation
parameter	void*	userdefined parameter, not used by PUB–Library

These operations work like the operations `bsp_reduce`, `bsp_scan`, `bsp_allreduce` and `bsp_allscan`, respectively. The difference is that these operations do not work on a continous set of processors (`group_lo`,..,`group_hi`) but they work on arbitrary processors which are specified by `dstlist` and `ndst`. `ndst` gives the number of processors which are involved and `dstlist` is an array of processor ids of length `ndst`.

Attention: The result is calculated asynchronously, so neither modify databuf nor use resultbuf before synchronization!

Global Operations

Global Operations

A global operation for bsp_reduce, bsp_lreduce, bsp_allreduce, bsp_lallreduce, bsp_scan, bsp_lscan, bsp_allscan, bsp_lallscan is defined by struct tbspop.

struct tbspop

member	**type**	**description**
operation	pbspop_function	function which calculates the operation (associative and commutative)
operandsize	int	size of one element

pbspop_function is declared as `typedef void (*pbspop_function) (void*, void*, int, void*)`. It is passed four arguments a, bAndResult, count, parameter.

parameter	type	description
a	void*	pointer to the first element of the left operand
bAndResult	void*	pointer to the first element of the right operand, result must be stored also here
count	int	number of elements
parameter	void*	user defined argument, not used by PUB–Library

The PUB–Library contains some standard operations:

operation	element type	description
bspop_addint	int	addition of integers
bspop_minint	int	calculate the minimum
bspop_maxint	int	calculate the maximum
bspop_minmaxint	int[2]	calculate minimum of first and maximum of second element
bspop_adddouble	double	addition of doubles
bspop_mindouble	double	calculate the minimum
bspop_maxdouble	double	calculate the maximum
bspop_minmaxdouble	double[2]	calculate minimum of first and maximum of second element

High Performance

bsp_createmsg

t_bspmsg* bsp_createmsg (t_bsp* bsp, int size)

parameter	**type**	**description**
bsp	t_bsp*	pointer to BSP object
size	int	total size of message including sizeof(t_bspmsg)

This function creates a message with at most size bytes for data and returns a pointer to the message. Call bspmsg_data to get a pointer to the data.

Information

bsp_b, bsp_g, bsp_l, bsp_s

int bsp_b (t_bsp* bsp)
double bsp_g (t_bsp* bsp)
double bsp_l (t_bsp* bsp)
double bsp_s (t_bsp* bsp)

parameter	type	description
bsp	t_bsp	pointer to BSP object

These functions return the BSP*–parameter B, g, L and s respectively. See `bsp_parameter` which return all parameters for more information.

bsp_parameter

void bsp_parameter (t_bsp* bsp, t_bspparameter* param)

This function fills the structure of type t_bspparameter param pointed to with the information about the machine the program runs on. More information on the BSP parameters l and g is available in [Val90] and on the additional BSP* parameter B in [BDM98].

struct tbspparameter

member	type	description
s	double	time for one cpu instruction in seconds, same as bsp_s
l	double	time for one syncronization in seconds, same as bsp_l
g	double	time for transmission of one byte, same as bsp_g
b	int	minimum size of a message for good performance, same as bsp_b
p	int	number of nodes, same as bsp_nprocs
computer	char[200]	name of machine
debug	int	not zero if PUB–Library is compiled in DEBUG–mode

Simulator

bsp_globalreg

void bsp_globalreg (t_bsp* bsp, void* data, int size)

parameter	type	description
bsp	t_bsp*	pointer to BSP object
data	void*	pointer to variable to register
size	int	size of variable in bytes

This functions registers a global variable. The simulator needs this because it uses threads with one global address space to simulate the nodes and each node must have its own instance of global variables.

References

[BDM98] Armin Bäumker, Wolfgang Dittrich, and Friedhelm Meyer auf der Heide. Truly Efficient Parallel Algorithms: 1-optimal Multisearch for an Extension of the BSP Model. *Theoretical Computer Science*, 203, 1998.

[BJvR98] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. Technical Report TR-RSFB-98-063, University of Paderborn, Computer Science Dept., 1998.

[GHL+96] Mark W. Goudreau, Jonathan M.D. Hill, Kevin Lang, Bill McColl, Satish B. Rao, Dan C. Stefanescu, Torsten Suel, and Thanasis Tsantilas. A Proposal for the BSP Worldwide Standard Library. Technical report, http://www.bsp-worldwide.org, April 1996.

[HDM97] J.M.D. Hill, S.R. Donaldson, and A. McEwan. *Installation and User Guide for the Oxford BSP Toolset (v1.3) Implementation of BSPlib.* Oxford University Computing Laboratory, November 1997.

[Val90] Leslie Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), August 1990.

Index

B

bsp_abort 16
bsp_allreduce . 24
bsp_allscan ... 24
bsp_b 28
bsp_createmsg 27
bsp_done 21
bsp_dup 21
bsp_findmsg .. 17
bsp_g 28
bsp_get 20
bsp_getmsg ... 17
bsp_globalreg . 29
bsp_gsend 23
bsp_gsendmsg 23
bsp_hpget 20
bsp_hpgsend .. 23
bsp_hplsend .. 23
bsp_hpput 19
bsp_hpsend ... 17
bsp_l 28
bsp_lallreduce 25
bsp_lallscan .. 25
bsp_lreduce ... 25
bsp_lscan 25
bsp_lsend 23
bsp_lsendmsg . 23
bsp_nmsgs 17
bsp_nprocs ... 15
bsp_oblsync .. 15
bsp_parameter 28
bsp_partition . 21
bsp_pid 15
bsp_pop_reg .. 19
bsp_push_reg . 19
bsp_put 19
bsp_reduce ... 24
bsp_s 28
bsp_scan 24
bsp_send 17
bsp_sendmsg . 17
bsp_sync 15
bsp_threadsafe_done 22
bsp_threadsafe_dup 22
bsp_time 15
bsplib_done .. 14
bsplib_init 14
bsplib_params_init 14
bspmsg_data . 18
bspmsg_size .. 18
bspmsg_src ... 18

F

functions
- description . 14
- list 12

G

Global Operations 25

S

simulator 29

T

t_bsplib_params . 14
tbspop 25
tbspparameter 28