Software Support for the BSP model

Oxford BSPlib Toolset

Introduction

The BSP model has been realized as a library of functions for process creation and destruction, remote memory access and message passing, and global, barrier-style, synchronization The BSP Toolset, BSPlib, that is being introduced in this document and used in the Programming Assignment allows a SPMD style of programming by offering two modes of communication, a message-passing approach and an one-sided direct remote memory access (DRMA) approach. When a software engineer writes an SPMD program that implements and uses a data-structure of some size n, then he creates p copies of the same program and splits the data-structure into p pieces of equal size so that copy i of the program maintains the i-th piece of size about n/p. The communication primitives of BSPlib allow the engineer to organize this data distribution efficiently and with the minimal effort by supplying him with those means (function calls) that would make data copying from or into the memory of a remote processor easy.

For the purpose of the programming assignment the DRMA approach will be introduced. The principal function calls that will be introduced are listed in the following table. As one physical processor (say, a workstation processor) is able to simulate more than one processors, we shall use the term “process” to refer to the processors of the BSP machine. Therefore one or more processes may be assigned to a single physical processor. Information that is available to a single process will be referred to as “local” whereas information that is available to all processes is referred to as “global”.

Table of Functions

Function Class	Function	Operation
Initialization and Termination	`bsp_begin`	Start of SPMD code
	`bsp_end`	End of SPMD code
Abnormal Stop	`bsp_abort`	One process halts all
Process Control	`bsp_nprocs`	Number of processes
	`bsp_pid`	Identifier of Calling Process
	`bsp_time`	Local (wall-clock) time
Synchronization	`bsp_sync`	Global Synchronization
DRMA	`bsp_push_reg`	Make memory info global
	`bsp_pop_reg`	Undo global effect
	`bsp_put`	Copy into remote memory
	`bsp_get`	Copy from remote memory
	`bsp_hpput`	High performance put
	`bsp_hpget`	High performance get

Initialization and Termination

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

Multiple processes from the same source are created by issuing the function `bsp_begin` and these processes are safely terminated by issuing a `bsp_end`. Only one pair of these statements can appear in a program and must be the first and last statements in a C function. The code fragment surrounded by these two statements will be executed in SPMD style on a number of processes. A more complex approach to process initialization utilizes `bsp_init` which is not described here. The integer argument `maxprocs` indicates the number of processes requested. If that many processes cannot be created, fewer will be. The actual number of processes spawn can be retrieved by issuing the call `bsp_nprocs()` which returns the actual number of processes. The simplest BSP program looks like the following one, `hello.c`, found in the `Examples` directory.

```
#include "bsp.h"
int main(void) {
  bsp_begin(bsp_nprocs());
  printf("Hello World from process %d of total %d\n",bsp_pid(), bsp_nprocs());
  bsp_end();
}
```

One can compile this BSP program as follows where instead of directly issuing the local C compiler, we call the BSPlib frontend that handles all relevant work for us. Compilation is no more complicated than the compilation of the corresponding sequential code.

```
% bspcc hello.c -o hello
```

Initialization and Termination II

BSPlib runs in three levels of efficiency. For the purpose of the programming assignment it is suggested that you use the default low-performance level (library level 0). If the most efficient implementation is used, the compilation line would look like the following one. In level 2, certain consistency checks in the BSP program are not performed.

```
% bspcc hello.c -flibrary-level 2 -o hello
```

We run program `hello` as follows.

```
% bsprun -npes 4 hello
```

The command `bsprun` runs a BSPlib linked program on a number of processes which are indicated as a parameter to *npes*. In our case 4 processes are requested. When `bsp_begin(bsp_nprocs())` is encountered the number of processes that appear as an argument to `npes` is the value returned by `bsp_nprocs()` and used as an argument to `bsp_begin`. This is an easy way a user can request a number of processes from the command line. After `bsp_begin` is executed subsequent calls to `bsp_nprocs()` return the actual number of processes which can be less than the number requested.

If BSPlib were installed on a four processor machine, in our example one process would be allocated on each processor. If it is a uniprocessor machine then all four processes will be allocated to the same processor. Function `bsp_pid` returns the identification number of the process issuing this call. The output of this program looks like the following one.

```
Hello World from process 3 of total 4
Hello World from process 1 of total 4
Hello World from process 0 of total 4
Hello World from process 2 of total 4
```

Initialization and Termination III

This simple program illustrates how BSPlib handles I/O. If a number of processes writes on the standard output/error, the write operations are performed in a non-deterministic way. The output of the `printf` statement (one per process) is printed on standard output in a totally arbitrary order. For types of operations other than output *(i.e. input, file access, redirection from standard output to a file, etc)*, the only guarantee is that process *zero (O)* will perform them correctly (the other processes *may* or *may not* work as expected).

Remarks

The total number of processes generated may be different than the argument to `bsp_begin`. The correct value is the one returned by `bsp_nprocs` after the `bsp_begin` call.

As there can be only one `bsp_begin` BSPlib supports static allocation of processors only. The state of the program before `bsp_begin` is inherited by process zero. No other process normally inherits this state. For example, arguments to `main` will be known to process zero only after `bsp_begin`. One needs to replicate them to the remaining processes.

For a BSPlib program to terminate gracefully, every allocated process MUST execute a `bsp_end`.

If intermediate files like `bsp21370_pre.c` and `bsp21370_pre.o` are generated, ignore them (by removing them).

`ATTENTION!`

As you are writing you first BSP programs make sure that no unterminated processes are hanging around (in case `bsp_begin, bsp_end` or `bsp_abort` were not issued properly). Check this by issuing, say, a

Abnormal Termination

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

Function `bsp_abort` provides a mechanism for safe and graceful error control and process termination under BSPlib. As a `bsp_begin` statements allocates more than one processes, in case of an error, these processes can be gracefully terminated if a single process issues a `bsp_abort` call. Parameter `format` is a C format string like the one used in `printf`. C program `abort.c` in the `Examples` directory in `bsp0` shows a use of this statement.

Remarks

If more than one processes issue a `bsp_abort` statement in the same BSP superstep, one, all or a subset of them may succeed in printing their format strings on standard error.

Process Control

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

Function `bsp_nprocs` returns the number of processes/processors that have been allocated if the call is issued before a `bsp_begin` and it returns the number of processes p allocated to the SPMD program if it is issued after a `bsp_begin (MAXPROC)`. In the latter case, it is $1 \leq p = bsp\_nprocs() \leq MAXPROC$. Each of p processes thus created is assigned a unique identifier id such that $0 \leq id < p$. This identifier is returned when a call `bsp_pid()` is subsequently issued. Examples that use these calls can be found in programs `control.c` and `nprocs.c` in the `Examples` directory.

Function `bsp_time` can be issued by any process id at any time after `bsp_begin` and the value returned is the time in seconds after `bsp_begin` and until this function call is issued by id. Even if processes issue time requests simultaneously the value returned by each one may be (and very likely is) different. Note that the time returned is wall-clock (NOT cpu) time. Program `bsptime.c` in the `Examples` directory is an example.

Remark

Note that `bsp_pid` returns an integer between 0 (inclusive) and *bsp_nprocs* (exclusive).

The time returned by `bsp_time` is wall-clock (NOT CPU) time.

Synchronization

`void bsp_sync(void);`

A BSP and a BSPlib program consists of a number of supersteps. Communication during a superstep becomes effective and remote data are guaranteed to be locally available at the end of that superstep. A function call `bsp_sync()` signifies the end of the current superstep (the segment of computation in BSPlib between two successive `bsp_sync` calls or between a `bsp_begin` and the first `bsp_sync`) and remote data are locally available after this call has been completed.

BSPlib supports only global processor synchronization in barrier style. This means that all processors MUST execute a `bsp_sync()` statement. Under library-level 0 a diagnostic will be printed (eg: ”Inconsistent supersteps between the processes”). Such a message means that processors are executing different `bsp_sync()` or perhaps one that appears in the same source line but in different iterations. Such a situtation may arise unde the following case

```
bsp_put(blah blah);

if (bsp_pid() == 10) {
   bsp_put(blah blah blah);
   bsp_sync();
}
bsp_put(blah blah);
bsp_sync();
```

Such a program fragment will raise such an error. Change it into something like.

```
bsp_put(blah blah);
if (bsp_pid() == 10) {
   bsp_put(blah blah blah);
   /* sync removed */
}
bsp_put(blah blah);
bsp_sync();
```

Direct Remote Memory Access

The DRMA communication facilities offered by BSPlib allow remote access of (communication into and from) any type of contiguous data structure including heap or stack allocated data. This is possible by allowing only preregistered memory areas to be accessible by remote processes. A memory area becomes available after being registered for remote access using a `bsp_push_reg` function call and ceases to be available after a `bsp_pop_reg` function call; both actions take effect in the following superstep. A `bsp_put` (or a more efficient `bsp_hpput`) call stores locally held data in the calling process into the registered memory area of a remote process. Similarly, a `bsp_get` (or a `bsp_hpget`) call fetches data from a registered memory area of a remote process into the local memory of the calling process.

Note that DRMA operations do not require the cooperation of the remote process and therefore data may be modified without the control of the remote process that stores them. Buffering can be used to increase the potential safety of these operations. There are various buffering schemes that are available.

buffered on destination, where writing into the registered areas will occur at the end of the superstep once all remote reads have been performed.

unbuffered on destination, where writing into the registered areas can take place at any time during a superstep (a semantically unsafe operation if the data-structures held in these areas are accessed during the superstep).

buffered on source, where information to be remotely communicated is copied first into a buffer and then transmitted.

unbuffered on source, where information is transmitted at any time during a superstep, a potentially unsafe operation if the data-structure stored in the registered area is changed during the superstep.

Note that in many other cases, such as when multiple processes write into the same registered memory locations, the information actually written is nondeterministically chosen.

Registration

```
void bsp_push_reg(const void *addr, int size);
void bsp_pushregister(const void *addr, int size);
void bsp_pop_reg(const void *addr);
void bsp_popregister(const void *addr);
```

Operations `bsp_push_reg` and (obsolete but still valid) `bsp_pushregister` are semantically identical. So are `bsp_pop_reg` and (obsolete but still valid) `bsp_popregister`. Parameter `addr` is the address of (first byte of) the area being registered or unregistered and `size` is the size of that area in bytes (i.e. a non-negative integer number).

Memory areas where information is going to be written into or copied from need to be registered in BSPlib. Each process thus issues a `bsp_push_reg` indicating the first address of a registered area and its size in bytes. Note that memory areas may be registered more than once; two areas can be registered for example with the same initial address but of different size.

Remark

Note that registration *takes effect at the following superstep*. A `bsp_sync` may be required after successive registrations. If a memory area is no longer required for remote communication it can be unregistered by issuing a `bsp_pop_reg` which also takes effect in the following superstep. If a BSP program tries to communicate into an *unregistered area*, a `runtime error` is *flagged*. The size of a registered area must be a non-negative number. Make sure in your programs that in each function each push corresponds to a pop. If a data structure has been pushed in another context pop it in that context as well so that you avoid errors.

Copy into remote memory

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

`pid` is the identifier of the process where data will be copied into.

`src` is the address of the first byte of the memory area that will be copied. The evaluation of src occurs in process `bsp_pid` issuing the function call.

`dst` is the address of the first byte where data will be stored (not necessarily copied) and must be preregistered.

`offset` is the offset in bytes from `dst` where `src` will start copying into. The calculation of offset is performed by the process that issues the function call. The reason such a parameter exists is that it allows us to copy into at addresses `dst+offset`, without the need to register all such addresses; only address `idst` needs to be registered and not all `dst+offset`.

`sze` is the number of bytes that will be copied from `src` into `dst+offset`. This assumes that `src` and `dst` are addresses of data structures of length at least `sze` and `sze+offset`, respectively.

The operation performed by a put (high performance or not) is similar to that performed by a `memcpy`. `bsp_put` is *buffered on source and destination*, whereas `bsp_hpput` is *unbuffered on source and destination*. Care must be taken when the high performance primitive is used.

The following program (`lshift.c` in `Examples`) shifts right the contents of variable x along the processes. Originally, the value of x at process i is i. After the remote communication the value of x in process i is communicated to the process whose identifier is one more than i (process $p-1$ sends its x value to 0).

Copy into remote memory II

```
#include "bsp.h"

int main(void) {
 int x;
 bsp_begin(bsp_nprocs());

 x=bsp_pid();
 printf("processor %d holds value %d before the put\n",bsp_pid(),x);
 bsp_push_reg(&x,sizeof(int));
 bsp_sync();

 bsp_put((bsp_pid()+1)%bsp_nprocs(),&x,&x,0,sizeof(int));
 bsp_sync();
 printf("processor %d holds value %d after the put\n",bsp_pid(),x);
 bsp_end();
}
```

The buffering of `bsp_put` ensures that data are read before being overwritten. If a `bsp_hpput` were used instead, it could have been the case that processor i would be sending to $i+1$ the value it received from processor $i-1$ and not its original value i.

Remarks

Only the destination area needs to be registered. The source area does not have to be registered. The destination area `dst` must be registered by a size which is at least `offset+sze`, otherwise a runtime error is flagged. `sze=0` communication does nothing. If `pid=bsp_pid`, a process copies into its own memory.

Copy from remote memory

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

`pid` is the identifier of the process where data will be copied from.

`src` is the address of the first byte of the preregistered memory area that will be copied. The evaluation of src occurs in process `pid`.

`offset` is the offset in bytes from `src` where data will be copied from. The calculation of offset is performed by the process that issues the function call.

`dst` is the address of the first byte where data will be stored (calculation occurs at the processor issuing the call).

`sze` is the number of bytes that will be copied from `src+offset` into `dst`. This assumes that `dst` and `src` are addresses of data structures of length at least `sze` and `sze+offset`, respectively.

Operation `bsp_get` is *buffered on source and destination*, whereas `bsp_hpget` is *unbuffered on source and destination*. Only the source `src` needs to be registered before the superstep in which the call is issued.

BSPlib Library

Copy from remote memory II

Remark

For buffered communication, all data related to put and get operations are read into buffers before the operations are effected (i.e. before the remote values are copied into the remote memory areas). For unbuffered communication, the copy operation can take place at any time during the superstep. The behavior of the library on one machine may be totally different from that on another machine.

The following program (`rshift.c` in `Examples`) shifts left the contents of x along the processors. Originally, the value of x at processor i is i. After the remote communication the value of x in processor i is communicated to the processor whose identifier is one less than i (process 0 sends its x value to $p-1$).

```
#include "bsp.h"
int main(void) {
 int x;
 bsp_begin(bsp_nprocs());
 x=bsp_pid();
 printf("processor %d holds value %d before the get\n",bsp_pid(),x);
 bsp_push_reg(&x,sizeof(int));
 bsp_sync();

 bsp_get((bsp_pid()+1)%bsp_nprocs(),&x,0,&x,sizeof(int));
 bsp_sync();
 printf("processor %d holds value %d after the get\n",bsp_pid(),x);
 bsp_end();
}
```

Remark Only the source memory area needs to be registered. The destination area does not have to be registered. The source area `src` must be registered by a size which is at least `offset+sze`, otherwise a runtime error is flagged. `sze=0` communication does nothing. If `pid=bsp_pid`, a process copies into its own memory.