A.V. Gerbessiotis
CIS 485/Spring 1999
PA 1

Programming Assignment 1
(Due: April 14, 1999)

1 What to hand in

You are asked to hand in the following deliverables.

(1) A `bsp1X.c` file, where `bsp1X` is your account, that contains all the C functions required below. Submit this file by e-mail to the address `alexg@cis.njit.edu` as an ASCII text file including the name `bsp1X.c` of the file in the Subject line of the e-mail. *Make sure* that your mailer DOES NOT send it as an attachment.

(2) An explanatory ASCII text document. Be brief and concise (i.e. for each function used explain what it does and what the various arguments denote). You may create this document from comments of the `bsp1X.c` file.

Make sure that the entry points of your functions adhere to the format outlined below or it will not be possible for them to be tested properly. As soon as `bsp1X.c` is received it will be compiled and linked to the testing functions.

Make sure that your source file compiles under standard ANSI C. Do not use extensions of ANSI C or any C++ constructs.

It is imperative that the file I receive from you DOES NOT include a `main()` function. Do your testing by linking this file to some separate testing files of yours. In case your C functions do not work as specified, you may receive partial credit depending on the documentation supplied (bug list etc).

For the remainder of this document the lecture notes on BSP and BSPlib (Handouts 7 and 8) and broascasting under the BSP model (Handout 9) will be used as a reference.

2 What to implement

Implementation is required for the function described in Part A. This part is worth 100 points.

Part B is *optional*. If you decide to solve this part, this can only enhance your grade especially if you did not perform well in previous problem sets/exams. If you decide not to do this problem, this will NOT affect your final grade in a negative way. This part is worth 75 extra points.

3 Which machine to use

Telnet in `logic.njit.edu` and use login names/passwords assigned in class. **NOTE:** This information may change when the Programming Assignment is handed out in class.

4 Part A: Broadcasting by k-ary replication

You are required to implement the broadcasting algorithm that uses k-ary replication and also to be presented in class.

You are asked to write a C function with the following syntax and behavior.

```
void bsp_mbroadcast(int fromp,
                    int multi,
                    int degree,
                    char *from,
                    char *to,
                    int nbytes);
```

- `fromp` is the source processor that holds the message to be broadcast. Note that in the algorithm presented in Handout 7 it is assumed `fromp=0`. You are now required to generalize that algorithm and make it work for any value of `fromp`.
- `multi` is the length of the data structure that is being copied. (the structure to be broadcast is an array of `multi` entries, the size of each entry being `nbytes` long).
- `degree` is the degree of replication. In other words, each processor, in each superstep, would send the received message to at most `degree-1` other processors.
- `from` is the base address of the first byte that will be broadcast.
- `to` is the base address of the first position that will receive the broadcast message.
- `nbytes` is the size in bytes of the elementary data type (array element) that is being copied.

Explanatory Instructions

(1) Make sure that this function works even if `from` and `to` point to the same memory location, i.e. `from=to`.

(2) If `degree` is less than two or greater than `bsp_nprocs()`, it should be set to `bsp_nprocs()`.

(3) You may assume that the remote memory area pointed to by `to` has been preregistered (it is the responsibility of the caller of this function to register the remote memory area).

Examples

1. Let us assume that variable `x` stored in processor 2 is to be broadcast to the remaining processors. Let the data-type of `x` be `double`. The following code fragment broadcasts `x`.

```
bsp_push_reg(&x,sizeof(double);
bsp_sync();
bsp_mbroadcast(2,1,degree,(char *)&x,(char *)&x,sizeof(double));
bsp_pop_reg(&x);
```

2. Let `y` of processor 0 be an array of `float`s to be copied into variable `z` of the same type, but which is statically allocated. Variable `y` will also be copied into a dynamically allocated memory area whose first position is pointed to by `zp`. The first element of `y` will be copied into `(zp+1)` and so on. Say that 20 floats will be broadcast. The following code fragment realizes both broadcast operations.

```
float y[20], z[20];
float *zp;

zp=(float *) malloc(sizeof(double)*21);

bsp_push_reg(z,20*sizeof(double);
bsp_push_reg(&zp[1],20*sizeof(double);
bsp_sync();

bsp_mbroadcast(0,20,degree,(char *)y,(char *)z,sizeof(double));
```

```
bsp_mbroadcast(0,20,degree,(char *)y,(char *)&zp[1],sizeof(double));

bsp_pop_reg(z);
bsp_pop_reg(&zp[1]);
```

In the examples above `degree` could take any positive value (i.e. we did not make explicit the degree of replication).

Remark
BSPlib function `bsp_bcast` also implements a broadcast operation. Its syntax is different from the one specified above for the Programming Assignment. It implements only a binary replication broadcasting algorithm. You are allowed and encouraged to view the source code of the function if you think it may help you.

5 Part B (Optional): 2-phase broadcasting

You are asked to implement the 2-phase broadcasting algorithm explained in Handout 7. Realize function `bsp_2_mbroadcast` below.

```
void bsp_2_mbroadcast(int fromp,
                      int multi,
                      char *from,
                      char *to
                      int nbytes);
```

This function utilizes the arguments of `bsp_mbroadcast`. If a data-structure of `multi` elements of an `nbytes` bytes long data-type is to be broadcast from processor `fromp`, then in the 2-phase algorithm, this processor would first split the `multi` elements into p pieces such that if $i < (multi \bmod p)$, then piece i will hold $\lceil multi/p \rceil$ elements, and if $i \geq (multi \bmod p)$, then piece i will hold $\lfloor multi/p \rfloor$ elements. In the first superstep, processor `fromp` sends the i-th piece to processor i, and in the following superstep, processor i replicates the i-th piece to the remaining processors.

Note that BSPlib function `bsp_bcast` also implements a variant of this algorithm. In that variant all pieces but the last one consist of $\lfloor multi/p \rfloor$ elements, whereas the last one consists of the remaining elements. This implementation does NOT constitute a solution for this part and no partial credit will be given for such or a similar solution. Your proposed solution *should split* the data-structure as evenly as possible. Although which processor is going to get $\lfloor multi/p \rfloor$ or $\lceil multi/p \rceil$ elements is not important it is important that in your proposed solution no processor receives more than $\lceil multi/p \rceil$. One way to achieve such a balance is the method described above. It is up to you whether you follow it or not.