A.V.Gerbessiotis
CIS 485/Spring 99
Programming Assignment 2

Programming Assignment 2
Due by NOON Wednesday, April 28, 1999

1 What to hand in

You are asked to hand in the following deliverables.

(1) A `bspX.c` file, where `bspX` 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 `bspX.c` of the file in the Subject line of the e-mail. *Make sure* that your mailer doesn't 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 `bspX.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 `bspX.c` is received it will be compiled and linked to my testing routines.

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 Handout 9 (Broadcasting on the BSP model) and the lecture notes will be used as a reference.

2 What to implement

Implementation is required of the function described in Part A. Part A is worth 100 points.

3 Which machine to use

Telnet in `logic.njit.edu` and use login names/passwords assigned in class.

4 Part A: 2-phase parallel prefix algorithm

You are asked to implement a 2-phase parallel prefix algorithm similar to the 2-phase broadcasting algorithm presented in class, where the associative operator operates pairwise to the elements of two vectors (not just two scalar values). If two vectors are combined by the operator, the result is a vector of the same size whose i-th element is the combination of the i-th elements of the two vectors.

The algorithm works similarly to the 2-phase broadcasting algorithm. Initially, each processor holds a vector of elements (scalar values) of some (vector) size. Each processor splits its own vector into p pieces and sends piece i to processor i. Processor i receives p i-th pieces (one from each processor). It then computes a sequential prefix problem on vectors, where the j-th vector is the i-th piece received by processor i from processor j. In the final phase of the algorithm the j-th result vector of the prefix operation at processor i is sent back to processor j which reconstructs from the received results the result of the parallel prefix.

Each vector consists of `multi` elements/components and each component is `nbytes` bytes long. The function that implements this algorithm will have the following prototype.

```
void bsp_2_mprefix(void (*operator)(),
                   int multi,
                   char *from,
                   char *to
                   int nbytes);
```

where

- `operator` is a pointer to a user-supplied function that takes four arguments and has the following prototype.

  `void operator(data *result,data *left,data *right,int size);`,

  where `result, left, right` point respectively to three arrays/vectors whose elements are of the elementary data-type `data` and each array is of length at least `size`. Function operator combines the i-th entry of `left` with the i-th entry of `right` and stores the result in the i-entry of `result` for all i such that $i < size$. For those of you familiar with the C standard library function `qsort`, the functionality (not semantics!!) of `operator` is similar to that of `compare` in that function.
- `multi` is the length of the input vector/array over which parallel prefix will be performed.
- `from` is the base address of the first element of the input array/vector F over which parallel prefix will be performed.
- `to` is the base address of the first element of the result vector T.
- `nbytes` is the size in bytes of a vector element/component.

Let F_i be the vector stored in processor i (i.e. abusing notation `from = (char *) &`$F_i[0]$). Let the result vector stored in processor i be denoted by T_i (i.e. `to = (char *) &`$T_i[0]$). At the completion of the algorithm, $T_i = F_0 + \ldots + F_i$, where $+$ denotes component-wise combination of two vectors that is for three vectors A, B, C of length n, $A = B + C$ is computed with the call `operator(A,B,C,n);`

Note that the operations performed in `bsp_2_mprefix` should be `data`-blind, that is, no reference to the `data` data-type should be made, except for its size which is the last argument of the function call.

Remark

Your implementation **should work for any vector size n (`multi` in our notation)**. This would require that the i-th piece of a vector to be of size $\lceil multi/p \rceil$ or $\lfloor multi/p \rfloor$ depending on whether i is less than or at least $(multi \bmod p)$.

Example

(1) Let us have two arrays `one, two` of a scalar C data-type that allows addition of such data-type instances. Let the length of each array be `size`. A parallel prefix operation will be issued on `one` and the result will be stored in `two`. The following code fragments realize this operation.

```
data *one,*two;

one= (data *) malloc(size*sizeof(data));
two= (data *) malloc(size*sizeof(data));

bsp_push_reg(one,size*sizeof(data));
bsp_push_reg(two,size*sizeof(data));
bsp_sync();
```

```
/* Input  vector one is set to some input values */

bsp_2_mprefix(oper_add,size,(char *)one, (char *) two,sizeof(data));
```

where `oper_add` has been defined as follows.

```
void oper_add(data *result, data* left, data *right, int n)
{
   int i;
   for(i=0;i<n;i++)
     result[i]=left[i]+right[i];
}
```

Note that `data` could have been defined in a definition file as follows.

```
typedef int data;
```

Remark

Note that BSPlib function `bsp_scan` implements a variant of this algorithm. In that variant, all but the last pieces of a vector have $\lfloor multi/p \rfloor$ elements and the last vector piece holds the remaining elements. This does NOT consitute a solution for this part nor WILL any credit be given to such a solution.

5 Part B (optional for Extra credit): Tree-based parallel prefix algorithm

Implement a parallel prefix algorithm that works on a k-ary tree (like the broadcasting algorithm in Part A of PA1) and derives from the $\lg p$-round PRAM algorithm presented in class (as opposed to the $2 \lg p$-round binary tree-based algorithm also presented in class). A prototype for this function is given below.

```
void bsp_mprefix(void (*operator)(),
                   int multi,
                   int degree,
                   char *from,
                   char *to
                   int nbytes);
```

where variable declarations follow the rules outlined in Part A of this assignment and Programming Assignment 1.