A.V.Gerbessiotis
CIS 485/Spring 99
PA 3

Programming Assignment 3
Due by NOON, Wednesday, May 5, 1999
LATE SUBMISSIONS WILL NOT BE ACCEPTED.

1 What to hand in

You are asked to hand in the following deliverables.

(1) A `bspXX.c` file, where `bspXX` is your account, that contains the C function described below. Submit this file by e-mail to the address `alexg@cis.njit.edu` as an ASCII text file including the name `bspXX.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 `bspXX.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 `bspXX.c` is received it will be compiled and linked to my testing routines (also provided in this assignment). In case your C function does not work as specified, you may receive partial credit depending on the documentation supplied (bug list etc).

For the remainder of this document Handout 11 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.

Note that you are not required to implement the function from scratch. You just need to fill in the pieces missing from the code provided below in textual format (with comments) and also available (directions follow in one of the following sections) as an ASCII `.c` file.

3 Which machine to use

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

4 What to get

In `logic.njit.edu` in directory `/home/bsp0/cis485` you may find a file `pa3-485.tar`. This file will also become available in the course web page. Transfer this file into your local account, untar it using the command `tar xvf pa3-485.tar` in a directory of yours. A number of files will be unfolded. One is a `Makefile` that allows you to compile your implemented code under BSPlib and link it with one of the test functions implemented by the instructor. Another file is `readme` that provides some information about the untarred files. File `matrix.h` includes definitions and declarations used by the other source files. C source code file `mult1.c` contains a `main()` function and debugging code that links with the function to be implemented and allows testing of it. File `general.c` is the C source template file that you need to fill in to complete this assignment. It partially implements the function whose full implementation is required for this assignment.

5 Part A: Synhcronization efficient matrix multiplication

You are asked to implement a matrix multiplication algorithm that is computation and synchronization efficient but not memory efficient. This is algorithm MULT_A of Handout 11. The full C source code for the following function is thus required.

```
void multiply_par(double *A,double *B,double *C,int n,int pi,int pj)
```

The distribution of A and B is identical to that describing the implementation of MULT_B in Handout 11. Initially, two global $n \times n$ matrices A_g and B_g are distributed among p processors. The p processors are divided into $\sqrt{p}$ groups of $\sqrt{p}$ processors each. Element (i,j) of A_g or B_g is stored in the $i/\sqrt{p}$-th processor of the $j/\sqrt{p}$ processor group, that is, processor $(j/\sqrt{p}) * \sqrt{p} + i/\sqrt{p}$. Function `multiply_par` requires six arguments, the two input matrices A and B which contain the n^2/p elements of A_g and B_g that are local to a particular processor, the result C that will hold n^2/p elements of the product $C_g = A_g \times B_g$, the dimension n, and pi, pj that for the purpose of this assignment should both be equal to $pi = pj = \sqrt{p}$ (which must be an integer).

As we have previously mentioned, A, B and C are ANSI-C pointers to a `double` data type. We store matrices in the form of an on dimensional array. This way, element (i,j) of a two-dimension matrix is stored in position $j*n+i$ of the one-dimensional array. All indices are in the range $0, \ldots, n-1$.

```
1. #include <stdio.h>
2. #include "bsp.h"
3. #include "matrix.h"

/* It implements a memory inefficient matrix multiplication routine,
 * on the BSP model suggested by Valiant (C.ACM 33(8),pp103-111,Aug 1990)
 * Initially, two global $n \times n$ matrices $A_g $ and $B_g$ are
 * distributed among  $p$ processors. The $p$ processors are divided
 * into $pj$ groups of $pi$ processors each.  Element $(i,j)$ of $A_g$
 * or $B_g$ is stored in the $i/pi$-th processor of the $j/pj$ processor
 * group, that is, processor $(j/pj)*pi + i/pi$.
 * Function {\tt multiply\_par} requires six arguments, the two input
 * (distributed among the processors) matrices $A$ and $B$ each
 * of dimension  (n/pi X n/pj), the result (already malloced)
 * of dimension (n/pi X n/pj), the dimension n, pi and pj.
 * $A$, $B$ and $C$ are ANSI-C pointers to a {\tt double} data type.
 * We store matrices in the form of an on dimensional array. This way,
 * element $(i,j)$ of a two-dimensional matrix is stored in position
 * $j*n+i$ of say, $A$. All indices are in the range $0, \ldots, n-1$.
 *
 * Each processor in one superstep reads the all the block it needs
 * to perform the matrix multiplication, and then it performs it.
 *  Size of memory per processor instead of 3n^2/p is
 *  4n^2 /p + 2n^2/q, where q=min{pi,pj}.
 *  All matrices are one-dimensional. Thus, element (i,j) of an
 *  nXn matrix X is the (n*j+i)-th element of X  X[n*j+i]; a column
 *  major order is used to store the matrix.
 *  Matrix A is transposed prior to communication and computation; this
 *  may increase efficiency due to locality - caching - issues.
 */
```

Function definition and variable declarations follow. Note that you are *allowed* to add new variables, but DO NOT delete the ones defined.

```
4. void multiply_par(double *A,double *B,double *C,int n,int pi,int pj)
{
5. register int nprocs=bsp_nprocs(); /* number of processors */
6. register int pid=bsp_pid();       /* who am i ? */
7. double  *a,*b,*tA;   /* aux space for A, B and transpose(A) */
8. register double tmp;      /* aux. variable in matrix mult.  */
9. register int  i,j,k,l;     /* Indices */
10.register int    t,        /* Auxiliary variable  */
11. ppi,ppj,        /* Index within and processor group of pid */
12. ni,nj,               /* A and B are n/pi X n/pj matrices */
13. si,sj,               /* Total size in bytes of a and b  */
14. p_b,        /* Base processor id. in communication  */
15. p_o,     /* Offset from base   in communication  */
16. a_b,     /* Base address of communicated data    */
17. t1,t2;      /* Indices/Addresses in multiplication  */
```

In the following piece of code various checks are performed. Variables `ni,nj` hold the number of lines and columns of A_g stored locally (i.e. in A and B). Given the processor identifier of `pid`, on lines 25 and 26, two other processor identifiers are computed: the processor group that a processor belongs to and its index within that group. For example if $pi = pj = 3$ and $p = 9$, processors $0, 1, 2$ belong to processor group 0, and processors $3, 4, 5$ to group 1 and so on.

```
  /* Dimension of A, B and C ; must be integer */
18.  ni=n/pi;
19.  nj=n/pj;
20.  check_if_valid((ni*pi),n,"multiply_par","n/pi must be integer");
21.  check_if_valid((nj*pj),n,"multiply_par","n/pj must be integer");

  /* nprocs must be equal to pi*pj or error */
22.  check_if_valid((pi*pj),nprocs,"multiply_par","nprocs != pi*pj");

  /* size of a and b (in bytes) respectively */
23.  si=ni*n*sizeof(double);
24.  sj=n*nj*sizeof(double);
25.  ppi=pid % pi;  /* index of pid within its (column block) group */
26.  ppj=pid / pi;  /* (column-block) group pid belongs to */
```

Allocation of matrices a and b where blocks of A and B from remote processors will be transferred into is performed in the following lines.

```
  /* Allocation and checking */
27.  a = (double *)malloc(si);
28.  check_if_null((void *)a,"in multiply_par","a");
29.  b = (double *)malloc(sj);
30.  check_if_null((void *)b,"in multiply_par","b");
31.  tA= (double *)malloc(ni*nj*sizeof(double));
32.  check_if_null((void *)tA,"in multiply_par","tA");

33.  bsp_pushregister(a,si);
```

```
34.  bsp_pushregister(b,sj);
35.  bsp_sync();
```

Matrix C (result) is initialized and matrix A is transposed as per the discussion of the implementation of MULT_B in Handout 11.

```
  /* Initialize result */
36.  for (j=0;j<nj;j++)
37.    for (i=0,t=j*ni;i<ni;i++)
38.      C[t + i]= 0.0;

   /* Transpose A first for efficiency */
39.    for (j=0,t1=0;j<ni;j++,t1+=nj)
40.        for (i=0,t2=0;i<nj;i++,t2+=ni)  {
41.     tA[t1+i] = A[t2+j];
        }
```

The missing lines 43-49 need to filled. This piece of code allows the reading of `pj` blocks of A into a. You may fill the code using fewer or more than the alotted lines.

```
  /* read pj  blocks of matrix A locally, as needed */
42.  for(l=0;l<pj;l++) {
         /* Implementation Point 1 */
43.
44.
45.
46.
47.
48.
49.
     }
51.   bsp_sync();
```

Fill in lines 53-59 below as well.

```
  /* Read pi blocks of matrix B locally,  as needed */
52.  for(l=0;l<pi;l++) {
/* Implementation Point 2 */
53.
54.
55.
56.
57.
58.
59.
   }
60.   bsp_sync();
```

Perform local matrix multiplication in lines 61-66. The three loops of matrix multiplication are shown incomplete.

```
   /* Do the multiplication locally */
/* Implementation Point 3 */
      for (j=0  ...
        for (i=0 ...
            for (k=0, ...
```

Matrix multiplication has been completed. Deregistration and deallocation of variables complete the function.

```
  /* De-registration */
  bsp_popregister(b) ;
  bsp_popregister(a) ;

  /* De-allocation */
  free((void *)tA) ;
  free((void *)b) ;
  free((void *)a) ;
}
```

Good Luck and Happy BSP Programming!