CIS 786: Programming Component : Homework 4/Problem 1 (100 points)

1 Introduction to MPI (30 points)

Do Homework (deliverable 2 only) in MPI.

- Use RMA operations only for communication (15 points).
- Use MessagePassing operations only for communication (15 points).

To distinguish your programs (and/or functions) for one or the other implementation add the suffix `_rma` or `_mpi` if the code pertains to the former or the latter case respectivly. Suffix `_rma` is to mean `remote memory access` and `_mpi` is to mean `message passing interface`.

Each user will participate with at most two entries: the `rma` and the `mpi` one. The winner entry is the one that satisfies two criteria: (a) best approximation of π, (b) best running time (wall-clock time). The instructor's decision is final; contestants cannot appeal! Solutions locked up by the instructor by Thu Nov 11, 2004 23:59:59EST can participate in the competition.

Note. 50 bonus points is the prize for the author of the winning implementation!

2 BSPlib programming (50 points)

The purpose of this assignment is dense matrix multiplication using BSPlib and DRMA communication primitives. You need to develop your own code to complete this assignment.

The following header file `alexmult.h` will contain some common definitions that will be used in this assignment. You are not allowed to interfere with this file (add or subtract information).

```
typedef float FTYPE;
```

You are going to multiply two $n \times n$ dense matrices named A and B and the result will be returned into a third matrix C of the same dimension. No assumption can be made on whether n is a multiple or not of p (number of processors). The two input matrices A and B will be given to you as one-dimensional column-major arrays (refer to Homework 2). For the purpose of this assignment n may vary up to 4096. Array C will be allocated and initialized by the calling function. The prototype of the function you are asked to implement is

```
void  mmcmi (FTYPE *A, FTYPE *B, FTYPE *C, int n , int p);
```

In `mmcmi`, `A,B` are the two input matrices, and C is the address of the output matrix. Parameter `n` is the dimension of A, B, and potentially of C i.e. each one will hold $n \times n$ keys. Parameter `p` is the number of processors that will be utilized for the multiplication. This however can be less than `bsp_nprocs()`. Therefore the context of calling this function could look like.

```
#include "alexmult.h"
 // This is a code fragment
#define N 1024
 FTYPE *A, *B, *C;
 int n,p,nprocs,pid;

 bsp_begin(bsp_nprocs());
 pid    = bsp_pid();
 nprocs = bsp_nprocs();
 n= N;
 p= nprocs;
 A = NULL ; B = NULL ; C = NULL;
 if (pid == 0) {
   A = (FTYPE *) malloc(n*n*sizeof(TYPE));
   if (A == NULL ) bsp_abort("Out of space : A\n");
   memset((void *)A, 0, n*n*sizeof(FTYPE)); // initialize with 0s
   B = (FTYPE *) malloc(n*n*sizeof(TYPE));
   if (B == NULL ) bsp_abort("Out of space : B\n");
   memset((void *)B, 0, n*n*sizeof(FTYPE)); // initialize with 0s
   C = (FTYPE *) malloc(n*n*sizeof(TYPE));
   if (C == NULL ) bsp_abort("Out of space : C\n");
   memset((void *)C, 0, n*n*sizeof(FTYPE)); // initialize with 0s
 }
 // Somewhere A, B are set up and initialized to some values.
 mmcmi(A,B,C,n,nprocs); //Multiply A x B , results in C.
 bsp_sync();
 // C is available ; print it sequentially somewhere here
 bsp_end();
```

Note that the input is available only in processor 0. Thus it is your responsibility to distribute portions of A and B into the remaining processors.

How this will be done, it's up to you. You may use any of the matrix distribution schemes we introduced in class. Which one you use will affect the performance of your algorithm and implementation (when you participate in the competition). As soon as you distribute A and B, your are allowed to disregard both A, B (i.e free memory). You are guaranteed that A and B will not be used again. Let us call the piece of A stored in processor i by A_i and do so similarly for B and C.

The expectation is each processor will get about n^2/p elements of A and B to start the multiplication. That is A_i and B_i should be $O(n^2/p)$ in size.

Whether for your parallel code you use column major or row major storage of the pieces is up to you. However, you must use one dimensional ANSI/ISO-C arrays only for A_i, B_i, C_i.

At the very end you need to communicate the result in array C of processor 0. This is the only processor that has allocated memory for C.

What determines your grade

- Effort (5 points).
- Correctness (5 points). List bugs if any, or assumptions made during the course of the design. Otherwise do not expect any credit here. Only correct (to the instructor) implementations will participate in the contest.
- Time efficiency (20 points). For a program that computes C correctly the parallel time will be compared to the performance of the sequential algorithm written by the instructor (uncommented code with no support will be provided as a tar file; it was used in Homework 2 for testing the code there). How many processors will be used for the testing will not be revealed. You may assume that it will be between 4 (inclusive) and 16 (inclusive). (Testing may occur on a homogeneous 16-machine, 32-cpu cluster). If the efficiency of your implementation is between 0 and 50 percent, you get 10 points, between 50 and 75 you get 15 points, 75 and 100 you get all 20 points and if your code gives superlinear speedups you can also collect 10 EXTRA points.
- Memory efficiency (20 points). You need a minimum of $3n^2/p$ memory for A_i, B_i, C_i. You receive 20 points if you use (per processor) less than $n^2/\sqrt{p}$ memory (unit of measure are FTYPE numbers). Beyond that you get only 10 points.

- Memory replication. If you replicate A, B to all processors you will be penalized. The penalty is going to be at least 25 points (subtracted from your grade).

Note. 50 bonus points is the prize for the author of the winning implementation in the contest!

3 Sequential LU decomposition with and without pivoting (20 points)

You may use as reference the electronic version of the book Numerical Recipes in C available at

`http://www.library.cornell.edu/nr/bookcpdf.html`

LU decomposition is described in Section 2.3 starting at page 43. You can use the code (**be reminded however, to quote in you source files that it is based on the Numerical Recipes in C book**) in this assignment as long as the product of your work conforms to the instructions given below. Implement the function

```
#include "alexmult.h"
int  lucm (FTYPE *A, int n)
```

that takes as input an $n \times n$ matrix A stored in an one-dimensional ANSI-C array A with element $A[i, j]$ stored in position $A[j * n + i]$. The output of `lucm` that will overwrite A is the LU-decomposition of A. The decomposition is in-place, i.e. A will be overwritten by the result. The return code of the function can be either -1 to signify normal termination (successful decomposition) or $0 \ldots n-1$ to signify abnormal termination (eg. decomposition failed). Then in this case the return code is the row/column that caused the failure. Since columns/rows are numbered starting from zero, this number is in the range $0 \ldots n-1$.

Note that the online book labels rows and columns starting from one not zero!

```
#include "alexmult.h"
int  lucmp(FTYPE *A, int n)
```

`lucmp` is the version of `lucm` that allows pivoting to be performed. The code returned from `lucmp` is -1 (normal termination) or 0 (abnormal termination).

If you read the book section carefully, and adjust its code twice (with and without pivoting) you can get 20 points total.

Note. 50 bonus points is the prize for the author of the winning implementation of `lucmp`! (Sorry, no prizes for `lucm`!