CIS 786: Programming Component : Homework 6/Problem 1 (100 points)

1 FFT Implementation (50 points)

Implement a parallel FFT. Information on FFT is available through the online notes, the textbook, the link `http://www.fftw.org/links.html` and the Numerical Recipes in C electronic book (Homework 4 link).

You are allowed to implement the function either in BSPlib using DRMA facilities or LAM-MPI using the MPI-2 RMA functions (put and get operations). Thus one of the following two function is to be implemented.

```
typedef double DTYPE;
void   fftbsp(DTYPE  *x, int n, int procs);

void   fftmpi(DTYPE  *x, int n, int procs);
```

The interface is simple: x is the input array of $2n$ complex numbers stored in pairs. The i-th pair is stored in $x[2*(i-1)]$ (real part) and $x[2*(i-1)+1]$ (imaginary part), where $i = 1, \dots, n$. The input is available in the x array of processor 0. You are responsible for distribution etc. The output should overwrite the input, i.e. stored in x in the same format. You may assume that n is a power of two. The maximum value of n can be assumed to be $n = 2^3 0$. The number of processors `procs` is the number of processors that can be used. You may assume that `procs` will be a power of 2.

What determines your grade

- Effort (30 points).
- Correctness (20 points). List bugs if any or assumptions made during the course of the design. Otherwise do not expect any credit here.

Note. 50 bonus points is the prize for the author of the fastest correct implementation.

2 Odd-even Merge Sort (50 points)

Sort n keys using a variant of odd-even merge sort on p processors. Assume n is a multiple of p. Thus n/p keys will be assigned per processor. In odd-even merge sort we have one key per comparator/processor. In the implementation n/p keys are assigned per processor. Whereas a comparator determines the minimum and the maximum of the inputs, two processors, that compares keys will determine the n/p smallest and n/p largest of the $2n/p$ they jointly hold. In the base case, the n/p keys per processor will be sorted using the `qsort` function available through the Standard C library. Thereafter, merging need only be performed.

```
 a -----  min(a,b)
     |
 b -----  max(a,b)
```

Implement

```
oddevensort(FTYPE *inseq, int n , int nprocs);
```

Initially, the input is available in `inseq` in processor 0. Split and distribute it accordingly. You may also assume that `n` is a multiple of `nprocs`.

You can implement the code in MPI (RMA) or BSPlib (DRMA).

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

3 Image Processing Operations (50 points)

The purpose of this homework assignment is to perform a number of image processing related operations in parallel. An image is an array of size `Ni x Mi` of pixels, A pixel is an unsigned integer type: it can be 8, 16, or 24 bits. Values for all these parameters are available through a file `aleximag.h` that may look like as follows. The actual values of `Ni, Mi` might be 256×256, or 512×512 or 1024×1024, and a `PIXEL` is an `unsigned short int`. For the purpose of this assignment we will also use 16×16 images as well.

```
#define Ni    512
#define Mi    512
#define Nmask 3
#define Mmask 3
typedef unsigned short int PIXEL;
typedef float FTYPE;
#define SMOOTHING 1
#define NOISE     2
#define HIGHPASS  4
#define PREWITT   8
#define SOBEL    16
#define LAPLACE  32
```

A mask is an `Nmask x Mmask` subarray. Each element of a mask has a weight w_i which in general is of some floating type `FTYPE`.

$$\begin{pmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{pmatrix}$$

One common image processing operation is the application of a mask on an image. This is to mean that the mask is applied to every pixel of the image. Consider what happens if we apply this operation to the pixel stored in row i and column j of an image `image1`. We can call this pixel $p_{i,j}$. If `image1` is stored in column major form in C array, then $p_{i,j}$ is stored in `image1[j*Ni+i]`.

$$\begin{pmatrix} p_{i-1,j-1} & p_{i-1,j} & p_{i-1,j+1} \\ p_{i,j-1} & p_{i,j} & p_{i,j+1} \\ p_{i+1,j-1} & p_{i+1,j} & p_{i+1,j+1} \end{pmatrix}$$

Then,

$$\begin{pmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{pmatrix} \times \begin{pmatrix} p_{i-1,j-1} & p_{i-1,j} & p_{i-1,j+1} \\ p_{i,j-1} & p_{i,j} & p_{i,j+1} \\ p_{i+1,j-1} & p_{i+1,j} & p_{i+1,j+1} \end{pmatrix} = \begin{pmatrix} - & - & - \\ - & p'_{i,j} & - \\ - & - & - \end{pmatrix}$$

where

$$p'_{i,j} = \frac{p_{i-1,j-1}w_1 + p_{i-1,j}w_2 + p_{i-1,j+1}w_3 + p_{i,j-1}w_4 + p_{i,j}w_5 + p_{i,j+1}w_6 + p_{i+1,j-1}w_7 + p_{i+1,j}w_8 + p_{i+1,j+1}w_9}{K}$$

with $p'_{i,j}$ being the new value of pixel $p_{i,j}$, and $K = \sum_i w_i$. That is, a mask changes the pixel value of the center pixel it operates on.

Some interesting masks are show below.

$$M_1 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \qquad M_2 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \qquad M_4 = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix},$$

M_1 is a smoothing mask, that takes the average value of the neighboring pixels. M_2 is a noise reduction mask, and M_4 is a high pass sharpening filter mask. Note that in the latter case $\sum_i w_i =$; we use $K = 9$ instead.

For edge-detection one can use

$$M_8 = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}, \qquad M_{16} = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}, \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}.$$

The Prewitt operator M_8 consists of two masks, and the Sobel operator M_{16} also consists of two masks. $K = 1$ in all cases. The result is the sum of the absolute values of the value obtained after applying the first of the two masks, and the value obtained after applying the second of the two masks. Since the results are independent the two operations can be done simultaneously.

A Laplace operator is M_{32} with $K = 1$.

$$M_{32} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}.$$

Implement first

```
imageproc(pixel *input, pixel ***output, int Ni, int Mi, unsigned short int MASK);
```

where `input` is an input-image of $Ni \times Mi$ size, and `MASK` is a mask that tell `imageproc` which masks to apply to the image. For example, if `MASK` is equal to 7, then this means that M_1, M_2, M_4 are to be applied since $1+2+4 = 7$. In general `MASK` is at most 63. The MASK in binary will be `0000 0000 0000 0111` for our example. Since in this example three masks/operators are to be applied, then three images are to be returned, one per mask. An image is an array of pixels, i.e. `pixel *image1`. An array of images is `pixel **ArrayOfImages`, i.e. `ArrayOfImages[i]` is an image. The second argument to `imageproc` is the address of an array of images i.e. `pixel ***`. This is because you will have to allocate space for output. You will have to figure-out the number of images in the output (that depends of `MASK`) and after that create, the space for each such image and pass the address of it to `output`.

After you solve the sequential problem, you are going to implement a simple parallelization of the sequential code. Split the image into p pieces, where p is the number of available processors. You may assume p is a power of two for this problem. Initially the image `input` is only available to processor 0. The result `output` should become also available to processor 0.

Your assignment is to distribute the image, split or in whole, to the p processors, split the computational load evenly among the processors, implement a parallelization of `imageproc`, and collect the final output in processor 0. You have got to implement `parimageproc`.

```
parimageproc(pixel *input, pixel ***output, int Ni, int Mi, unsigned short int MASK);
```

The problem with a parallelization of image processing functions is the operations performed on the boundary (perimeter) of an image. If pixel $p_{i,j}$ is on the perimeter of the image stored in processor 1 for example, then the its northern, southern, eastern, or western neighbor pixel might be stored non-locally, to a neighboring processor. Make sure you deal with these issues correctly.

You can use LAM MPI or BSPlib for this problem; the choice is yours.

```
 // How to allocate and return dynamically allocated memory
                                                     Memory            Meaning
                                                Address  Contents
 int  *A;                                           100 [001232438]    &A  is 100

 A= NULL;                                           100 [000000000]    A initialized to NULL
 allocate_memory ( & A ); // Pass the address of A  i.e. 100

void allocate_memory (int ** a) {
 int *C                                             500 [0230210212]   &C is 500

 C= (int *) malloc(10*sizeof(int));                 500 [0000000020]   Starting at address
                                                                       20, 10 integers are
                                                                       allocated.
 *a = C;                                            100 [0000000020]   Pass C to A.
}
```

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

4 Option A (50 points)

Propose by Nov 18, one implementation of similar difficulty to Problems 1-3 of your choice.

5 Option B (50 points)

Propose by Nov 18, one implementation of similar difficulty to Problems 1-3 of your choice.