ECE 473 --Assignment 04
Due Date Specified on BlackBoard
Global Sum II

Task: Write a parallel program and associated helper executables, according to the exact functionality given below, to open a binary file with integers in it, and find the sum of all the numbers.

In the parallel version, you will use a block decomposition so that each processor will be find a partial sum of its portion of the file. Then, the final summation of the partial sums will be done using our global_sum() function from the last assignment.

In order to carry out this objective, you'll need some helper executables:

```
make-list -n 100 -o outfile.dat
```

This serial executable will generate the initial data file that will contain the list of integers to be added. It takes two arguments "-n" that indicates the number of integers to place in the list, and "-o" to indicate the name of the datafile to write the integers to. If one of the arguments is not specified at runtime, there should be defaults with n=100 and the outfile called "default-make-list-file.dat". The first integer in the file should be the number of integers being written to the list. Then directly following this, all the other n numbers should go back to back. (File should be written as binary, NOT ASCII).

```
serial-add-list -i infile.dat
```

This serial executable will open the specified input file, and will determine the sum of all the numbers in it. It will assume the file format is equal to that of above. It should also have a default infile should the -o not be specified. This default should be the same as specified above. This is essentially the non-parallel version of the parallel one described further down in this assignment.

```
print-list -i infile.dat
```

This serial executable will open the specified input file and will print the contents to the screen in one long column. Again, defaults as specified above.

```
parallel-add-list -i infile.dat
```

This is the parallel version of the list adder. Defaults as specified above. As mentioned above, it will use block decomposition. You will likely find useful the author's utility files from Appendix B of the textbook. These two files, MyMPI.c and .h can be found on BlackBoard.

These file may have some places that will show warnings when compiled with modern compilers, as was the case on my laptop. You will need to correct these issues. Be careful of what you change here, you don't want some unintended side-effect. For this

project, you will likely find useful this following functions from those files:

```
void read_block_vector(char *, void **, MPI_Datatype,
int *, MPI_Comm);

void print-block-vector(void *, MPI_Datatype, int,
MPI_Comm);
```

and the macros from the MyMPI.h file, in particular, the:

```
BLOCK_SIZE(id,p,n)
```

The read_block_vector() function above, will open a file, read the first integer out of the file, and then will have each process malloc() the appropriate amount of space to hold it's part of the file. Then, one process will read the chunks of the file out, and send it to all the other processes. Essentially, this function does some very heavy lifting for you. The same effect can be achieved easily with MPIIO, something that we will discuss later. I expect you to look at what these functions to in order to determine what they do, how the do it, and ultimately how to use them to accomplish your goal.

Your project must be in a directory with the same naming convention as previous projects, and you must make a tar file to submit it as before. Your project MUST consist of the following files, nothing more, nothing less:

```
functions.c
make-list.c
MyMPI.c
parallel-add-list.c
print-list.c
serial-add-list.c
functions.h
MyMPI.h
Makefile
```

In your 'functions.c' file, you will write the functions that are used in, and in some cases, common to your .c files above.

```
void global_sum(double* result, int rank, int size, double
my_value);
// This is from the previous assignment
void make_list(int n, int **A);
//  This creates a list using an integer array that is dynamically allocated inside this function, and returns it via the argument A. In this example, you could just have A[i] = i;
void write_list(char* out, int n, int* A);
//  Takes a character array that is a filename and opens it, write n to the first location in the file, and then writes n integers from A into it ALL IN ONE fwrite().
void print_list(int n, int* A);
```

```
// Prints to stdout the contents of array A.

void read_list(char* out, int *n, int** A);
// Opens a file, and reads to contents out of it and returns this in an array via the
argument A. Read the contents of the file after the first location ALL IN ONE fread().
You'll be malloc'ing an array.
```

Having these functions located in that .c file means that the actual programs themselves are fairly simple, and just use the functions you create here. It also means you must support this interface and functionality so that I could write my own programs that use them, and that they would still work. For example, in my print-list.c file, the main part after all the initial error checking and argument parsing, all I do is:

```
int *A;
char *in ....
int n;
....
read_list(in, &n, &A);
print_list(n, A);
```

As I leverage the calls to functions defined in functions.c.

Use a Makefile that compiles the entire project, and names the executables as specified above.

Performance Evaluation:
I expect you to instrument your code in order to determine how long it takes to run. In an MPI program, you should use

```
MPI_Wtime()
```

as described in section 4.6 of the Quinn textbook. You should ensure that the processes are synchronized as described in that section by making use of

```
MPI_Barrier()
```

For your serial-add-list executable, use time() from time.h.

For each version (both parallel-add-list and serial-add-list), I want two times reported via STDOUT at program termination: The ENTIRE execution time of the program, and just the time required to do the addition. In other words, the entire elapsed time will include the rather costly fileIO, while the other one will encapsulate just the computation (and IPC in the parallel case). I want a report generated that will be included with the submission. This report will contain two types of graphs. Ones with the number of processors on the x axis, and execution times on the y axis, and ones with number of processors on the x axis, and speedups on the Y axis. (Use the definition of speedup as defined in the Quinn textbook in chapter 7). The first type will look like Figure 4.6 and the latter will look like Figure 7.2 (both from the Quinn Textbook). Also, like some of the slides we've used this semester.

For each x datapoint, I want you to get the timings for lists of increasing size. Therefore,

each plot will have multiple curves representing timings for different sized lists. This will show how the execution time of the parallel program changes as you increase the number of processors and increase the so-called 'problem-size'. The times you get need to be large enough that we're talking about at least a few seconds if possible, if not more; however be careful about the file size necessary to actually get these timings for the 'inner kernel'. All timings need to be done on Palmetto, even the sequential one using a PBS script.

Submission into BlackBoard:

I expect that your project will reside in a single directory, and that will have all necessary files in that directory. That directory should be named first_last_ass5, with the files named as mentioned above (in this case, all you need to have with your name embedded in it is the directory name.) From there, you will tar and gzip that directory, and submit that to BlackBoard. That can be done from the commandline as described in the previous assignment with a tar command piped to gzip.

Grading Rubric:
1) you will get minimum points for a completely working solution that exactly matches the specifications above.
"C"
2) you will get additional points if your sequential programs are error free in a 'valgrind' verification.
"B"
2) you will get additional points for a solution that meets the two above items and is also IMMACULATE, as I described in class, that you check all the error conditions of system calls you make, that you error check and validate all inputs from users.
"A"

Appendix:

Installing Valgrind.

Although you can likely obtain Valgrind from the OS package manager, I prefer that you obtain Version 3.6.0 it directly from:
http://www.valgrind.org/downloads/current.html
After you unzip it in a temporary location, cd to the directory, and type the following at the prompt.

```
$ ./configure
$ make
$ sudo make install
```

Then, you can run valgrind on your program by simply typing the following:

```
$ valgrind ./make-list -n 100 -o values.dat
```

Valgrind will then run your program, and check for errors particuarly associated with memory

bugs. Make sure you have 0 errors, and 0 leaked memory (or any type).
Although you can not run your parallel program in valgrind, you can at least do it for all the other helper executables, etc.

A tutorial I once wrote about how to use Valgrind is located here, but was from an old version of valgrind, use at your own risk:

http://www.parl.clemson.edu/~wjones/summerStudents/memoryDebugging/

Parsing Command Arguments

The function you want to use is "getopt()". Read the man page on this carefully. Here is a skeleton where I used it:

```
int num
char *ofile
while((opt = getopt(argc, argv, "n:o:")) != -1)
{
      switch(opt)
      {
            case 'n':
                  num = atoi(optarg);
                  ...
                  break;
            case 'o':
                   ofile = strdup(optarg);
                  ...
                  break;
      }
}
```