CSCI 473 --Assignment 5 – Fall 2019

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, it should print out a usage message indicating what should be done. 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. If no parameter is specified, it should print out a usage statement. 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. As mentioned above, it will use block decomposition. You will find MyMPI.c and MyMPI.h, which has some utilities that you may find useful for this assignment. They are included in the ZIP file that contains this assignment. Here are some that might be useful to you for the parallel version of the helper utilities.

```
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, of using the MyMPI.c and MyMPI.h, included in the zip file.

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.

You 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
```

Start with the GREEN functions / files / programs. They are all 'normal' in the sense that they are no parallel. You should be able to run a workflow like this first:

```
$ ./make-list -n 10 -o test1.dat

$ ls -la ./test1.dat
-rw-r--r--  1 wjones  1334764962    44B Oct  6
10:47 ./test1.dat

$ ./print-list -i ./test1.dat
0
1
2
3
4
5
6
7
8
```

```
9

Sum of items in list is: 45

$ ./serial-add-list -i ./test1.dat

Sum of items in list is: 45
Time is 0 seconds
$
```

Notice, we generated a file containing integers (10 of them), and the file size is 44 bytes, highlighted above. That's because the first integer was the number of elements, and then it was the data itself. As you can see, if we HEXDUMP the file, this is what we see:

```
$ hexdump ./test1.dat
0000000 0a 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00
0000010 03 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00
0000020 07 00 00 00 08 00 00 00 09 00 00 00
000002c
```

If you look at this, you can see the data in there that was print to the screen with the helper utility above. (10, 0, 1, 2, …). Remember, the data in the file is stored in binary, not ascii text. Read about this when you look at the man pages for fwrite() and fread().

My Recommendation

Start just with the print-list utility, using the example test1.dat file that I've posted with the moodle assignment, that would be a good place to start, rather than writing your own data out first. That way, you know you'll match what I expect. Then you can try to replicate it.

In your 'functions.c' file, you will write the functions that are used in, and in some cases, common to your .c files above. Your functions must have this exact specification / interface.

```
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 our textbook / the MPI standard. 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. You should also perform an global minimum reduction as in the author's textbook example for taking the timing of an MPI program.

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).

Submission into Moodle:

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 as in previous assignments, and submit that to Moodle. 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 (if you're running this on your own Linux distribution), or you could download and compile from source.

The CI project already has it installed:

```
[wjones@ciapp5 473]$ which valgrind
/usr/bin/valgrind
[wjones@ciapp5 473]$ valgrind --version
valgrind-3.14.0
[wjones@ciapp5 473]$
```

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 particularly 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 (summer 2003) about how to use Valgrind. It, along with the examples to which it refers is in Moodle, but was from an old version of valgrind, use at your own risk. I'm sure there are newer tutorials out there.

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:

```
while((opt = getopt(argc, argv, "n:o:")) != -1) {
     switch(opt) {
        case 'n':
           ...
           break;
        case 'o':
           ...
           break;
     }
}
```