Programming Assignment Multi-Threading and Debugging 2

Due Date:  Friday, March 8 @ 11:59 pm

PAMT2 Assignment Overview
The purpose of this mini-assignment is to continue your introduction to parallel programming and multi-threading with OpenMP. It will cover aspects of implementation, as well as the benefits and tradeoffs involved. You will again be provided with skeleton code, and this time you will fill in the functionality where necessary to determine whether a large number is prime or not by calculating the number of factors the number has. This will be done both sequentially in one thread and in parallel over many threads simultaneously using OpenMP directives similar to the first multithreading assignment. The program will also record and display the execution time of the calculations using both methods, allowing you to see the relative performance of each.

NOTE: Once again, due to CPU-intensive nature of this multi-threading assignment, we are developing and testing the multi-threaded programming assignment on the workstations (preferred) or ieng6. You can also develop on the pi-cluster, but the execution times will be much slower.

Debug2 Assignment Overview
You will also get to practice your debugging skills again. We have written a program that will start a command line tic-tac-toe game. We think it's close to working, but we didn't have time to debug it. It's now your job to help us track down the bugs and fix them so we can get it to work. You should reference your notes on static vs. global to help you finish this assignment. The debugging exercise must be completed on pi-cluster.

To summarize:
1. PAMT2 must be developed, executed, and turned in with your cs30x account on the workstations in the lab (preferred) or ieng6.ucsd.edu or pi-cluster remotely.
2. Debug2 must be debugged, executed, and turned in with your cs30x account on pi-cluster.ucsd.edu.

PAMT2 Grading
- **README: 10 points** - See PAMT2 README File section.
- **Compiling: 10 points** - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Correctness: 40 points** - Includes both abnormal and normal output, both to the correct destination (stderr vs stdout).

Debug2 Grading
- **README: 30 points** - See Debug2 README File section.
- **Correctness: 10 points**

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.
PAMT2 Overview
Once again, all functions for pamt2 are written in C. Your job is to fill in the missing functionality.

Gathering Starter Files for PAMT2:
$ cd ~
$ mkdir pamt2
$ cp ~/../public/pamt2StarterFiles/* ~/pamt2/

You should now have a directory named pamt2. This will contain the starter files for the project, listed below. You are responsible for filling in the code in function1.c and omp_function1.c.

<table>
<thead>
<tr>
<th>Starter files provided:</th>
</tr>
</thead>
<tbody>
<tr>
<td>function1.c</td>
</tr>
<tr>
<td>omp_function1.c</td>
</tr>
<tr>
<td>pamt2.h</td>
</tr>
<tr>
<td>Makefile</td>
</tr>
<tr>
<td>main.c</td>
</tr>
</tbody>
</table>

PAMT2 Sample Output
This program takes a list of long long integers (64 bit numbers, up to about 9 quintillion in decimal) and displays information on how many factors each number has, whether each number is prime and how long it took to determine that.

Below are a few examples on one of the B240 workstation (bold indicates user input):

[cs30xzz@its-cseb240-05]:pamt2$ ./pamt2 8081 246813607
Calculating number of factors for 8081 with sequential function1()
   [be patient with large values]
Number of factors: 2
8081 is prime
Completed in 0.000073 sec

Calculating number of factors for 8081 with OpenMP function1()
   [be less patient]
Num of threads = 8
Number of factors: 2
8081 is prime
Completed in 0.000039 sec

*** function1 Speed-up: 1.864738 ***

Calculating number of factors for 246813607 with sequential function1()
   [be patient with large values]
Number of factors: 2
246813607 is prime
Completed in 1.988375 sec

Calculating number of factors for 246813607 with OpenMP function1()
   [be less patient]
Num of threads = 8
Number of factors: 2
246813607 is prime
Completed in 0.399927 sec

*** function1 Speed-up: 4.971840 ***
Note that the first group of results for each number is for sequential mode (checking each possible factor of the number in sequence in a single thread), while the second group of results is for parallel mode using an OpenMP directive on the loop - review how to use OpenMP parallel directives from your pamt1 assignment and write-up.

Like last time, the speedup is the factor by which parallel computation of the result is faster than sequential. A speedup of 1 would mean that both methods are about equally fast, while a speedup of less than 1 would indicate that the sequential mode is better (the overhead of forking and joining threads can dominate small
problems), and a speedup of greater than 1 would indicate that the parallel mode is better. Your elapsed times and speed-ups might not be exactly the same as the sample output, but it should look reasonable and justifiable. They will definitely be different depending on which system you run your code on (lab workstation, ieng6 server, pi-cluster, etc.).

The number of factors calculated need to be the same between the sequential and parallel versions for any given number and match those of the sample executable. By most definitions, prime numbers are natural numbers greater than 1 that have no positive divisors other than 1 and itself. So negative numbers input will result in 0 factors and are not prime.

Sample executables: `~/.public/pamt2test` (for workstations and ieng6 servers)
`~/.public/pamt2test_pi-cluster` (for pi-cluster)

C routines:
```c
int main( int argc, char * argv[] );
long long function1( long long n, long long lo, long long hi );
long long omp_function1( long long n, long long lo, long long hi );
```

PAMT2 Modules

`main.c`
```c
int main( int argc, char * argv[] );
The driver of the program. For each number entered on the command line, main() calls gettimeofday() to get the start time, and then calls function1() to run the calculations sequentially. Upon return, gettimeofday() is called again to get the end time. The time difference is then calculated and the results of the sequential run are printed to stdout. Everything up to this point is given to you in the starter code.

It then does the same thing with omp_function1().

Note: There is a comment marked TODO in main.c that shows you where you should make additions/changes. You do not need to (and in fact should not) make changes to main.c anywhere except at this point. You should remove the return 0 once omp_function1() has been implemented.
```

`function1.c`
```c
long long function1( long long n, long long lo, long long hi );
To implement function1(), use the algorithm provided in the header.
```

`omp_function1.c`
```c
long long omp_function1( long long n, long long lo, long long hi );
To implement omp_function1(), copy over your working sequential code from function1.c. Then using what you learned in pamt1, create an OpenMP pragma to parallelize the loop you created for the third part of the algorithm. Remember to comment out the return 0 around line 78 in main.c when you are ready to test your omp_function1.
```

PAMT2 README File
Along with your source code, you will be turning in a README. Remember to follow all of the guidelines outlined in the README Guidelines.

Questions to Answer in the PAMT2 README
1. Try running pamt2 on a fairly large prime number 4294967291 and a fairly large non-prime number 4294967292. You should see that the sequential version takes much longer time than the parallel version. Try running pamt2 on a fairly small prime number 89 and a fairly small non-prime number 88. You should see that the sequential version takes much shorter time than the parallel version. For this program, does it matter if the number entered is prime or non-prime as far as the running time of the program is concerned? Why?

2. Try running pamt2 with a fairly large number (like 429496777) on one of the workstations, on ieng6, and recompile (make clean; make) and run on one of the pi-cluster nodes. Which is more important in general: 1) the overall speed-up gained by parallelizing or 2) the number of threads available to parallelize or 3) the time it takes to complete the sequential or parallel versions? Why? [Warning! Large values can take a fairly long time on a pi-cluster node.]

3. At about what value is the break-even point - where generally values larger than this break-even point the parallel version is faster on one of the workstations, on one of the ieng6 servers, and on a pi-cluster node?

Debugging Exercise 2 Overview
The purpose of this program is to play tic-tac-toe from the command line. The main() function is written in C, and it will run the game using various helper functions.

We provide all of the code to you, but the code doesn't quite compile or work as it should. It is up to you to track down the bugs and fix them. There are a total of 8 bugs (2 compile-time bugs, 6 run-time bugs) in the source code.

You are required to record ALL of the bugs we placed and the solution for each bug. See the section on Debug2 README File for more details.

You **DO NOT** need to modify any file/function headers.

Gathering Starter Files for Debug2:
For debug2 (the debugging exercise), you will develop on pi-cluster as you do for most programming assignments. However, we will provide you with the buggy code. Simply go to your home directory and copy the whole folder to your home directory:

```bash
$ cd ~
$ cp -r ~/../public/debug2 .
```

This will provide you with all of the buggy source code. All you have to do is fix the code and detail in a README file exactly what fixes you had to make to get this code to work properly. Include:

- **Effects** of the bug. What signaled to you that this bug exists? This can be error messages for compiler errors, weird behavior for runtime errors, etc.
- **The line number(s)** of the fix(es). (These may change as you fix more bugs, so it's fine if the numbers aren't exact)
What the line(s) looked like before and after your fix(es).

A short explanation (1-2 sentences is fine) of your reasoning behind each fix and how you debugged the problem.

Keep in mind that each bug fix should only modify at most three (3) lines of code. These lines may be separated by other lines, but if you can logically consider them as part of the same error, they should be documented together. However, if you find yourself thinking of a fix that changes more than this number of lines, either the bug is not in those lines, or your fix can be written more concisely.

Another thing to note (if you aren't already doing this): if, while trying to fix a bug, you notice some code is definitely wrong but fixing it does not get rid of the bug -- make note of your fix for later, but work on solving the current bug first. Since you need to document the effects of each bug, you should undo this early fix until you find the actual fix to the current bug you are encountering.

Debug2 Sample Output
The program takes no arguments from the command line:

```bash
$ ./debug2

Below are a few examples (bold indicates user input):

If the user didn't pass in the correct format, an error message is printed.
[cs30xzzz@ieng9]:debug2$ ./debug2
-------
| | | | 
-------
| | | | 
-------
| | | | 
-------
Player 1, please enter your move: (1,2)

Missing right paren
Player 1, please enter your move: ^D

As you can see, the program will continue running until the game is over. If player 1 or player 2 wins, or there is a tie, the game will end.

[cs30xzzz@ieng9]:debug2$ ./debug2
-------
| | | | 
-------
| | | | 
-------
| | | | 
-------
| | | |
-------
Player 1, please enter your move: (0,0)
Player 2, please enter your move: (1,0)

Player 1, please enter your move: (0,1)

Player 2, please enter your move: (1,1)

Player 1, please enter your move: (0,2)

Player 1 Won!

If the game is completed without a winner, a tie message will be printed.
Player 1, please enter your move: (0,0)

<table>
<thead>
<tr>
<th>X</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Player 2, please enter your move: (0,1)

<table>
<thead>
<tr>
<th>X</th>
<th>O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Player 1, please enter your move: (1,0)

<table>
<thead>
<tr>
<th>X</th>
<th>O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Player 2, please enter your move: (1,1)

<table>
<thead>
<tr>
<th>X</th>
<th>O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Player 1, please enter your move: (2,1)

<table>
<thead>
<tr>
<th>X</th>
<th>O</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Player 2, please enter your move: (2,0)

---------
| X | O |   |
---------
| X | O |   |
---------
| O | X |   |

Player 1, please enter your move: (0,2)

---------
| X | O | X |
---------
| X | O | O |
---------
| O | X |   |

Player 2, please enter your move: (1,2)

---------
| X | O | X |
---------
| X | O | O |
---------
| O | X | X |

Player 1, please enter your move: (2,2)

---------
| X | O | X |
---------
| X | O | O |
---------
| O | X | X |

It’s a tie!

C routines:
int main( int argc, char * argv[] );
long parsePlay( long board[][3], char * input, long * xIndex, long * yIndex );
long checkWin( long xIndex, long yIndex, long turn );
void printBoard( long board[][3] );

Assembly routines:
long checkMove( long board[][3], long xIndex, long yIndex );
Debug2 C Modules

debg2.c
int main( int argc, char * argv[] );

The main driver of this program. It combines all of the game's logic as well as controlling the players and the placement of the pieces. It also contains the main game loop.

Return Value: EXIT_SUCCESS.

parsePlay.c
long parsePlay( long board[][3], char * input, long * xIndex, long * yIndex );

This function parses the user input from stdin. The input must be of the form (X, Y). This function uses strchr() and strtol() to parse the X and Y arguments one at a time. It also replaces the comma and right paren with null characters to allow strtol() to parse X and Y.

Reasons for error:
• Input isn't of the form (X, Y).

Return Value: 1 if the argument was successfully parsed, or 0 on failure.

checkWin.c
long checkWin( long xIndex, long yIndex, long turn );

This function checks whether or not a player has won the game. It keeps track of the number of times that each player has played in each row, column, and diagonal. It increments the count if player 1 plays, and decrements the count if player 2 plays. If any row, column, or diagonal adds up to 3 or -3, then the player has won.

Return Value: 1 if a player has won, or 0 otherwise.

printBoard.c
void printBoard( long board[][3] );

This function simply prints a formatted board to stdout. 'X' represents player 1, 'O' represents player 2, and ' ' represents empty.

Return Value: None.

Debug2 Assembly Modules

checkMove.s
long checkMove( long board[][3], long xIndex, long yIndex );
This function checks if $\text{board}[\text{xIndex}][\text{yIndex}]$ is a valid position. It first checks if the indices are in range, and then checks if the position has been filled yet.

Return Value: 0 if the position is not valid, 1 if true.

**Debug2 README File**

For the debugging assignments only, you do not have to include the usual high level description, how tested, etc. in your README file. You will, however, have to list the compilation error you encountered and the fix you made to correct the error.

You will also have to solve several logic errors. Again, for each problem, describe:

- **Effects** of the bug. What signaled to you that this bug exists? This can be error messages for compiler errors, weird behavior for runtime errors, etc.
- The **line number(s)** of the fix(es). (These may change as you fix more bugs, so it's fine if the numbers aren't exact)
- **What the line(s) looked like before** and **after** your fix(es).
- A **short explanation** (1-2 sentences is fine) of your reasoning behind each fix and how you debugged the problem.

As a guideline, there should be 2 compilation errors and 6 functionality problems. Make sure you locate all of them! (Note: When we say there is 1 compilation error, we mean that there is one fix you'll have to make, not that there is one error printed to the screen. In general, you should not have to modify more than 3 lines of code per fix).

If you happen to find more than 8 errors, document all of them anyway. It is most likely that you are considering one error as many different errors. We will grade you based on what fixes you make - not the number of errors you find. In other words, you will not lose points for documenting extra errors.

**Turn-in Instructions**

**Complete Turnin - due Friday night, March 8 @ 11:59 pm**

Once you have checked your output, compiled, executed your code, and finished your README files, you are ready to turn in your assignments.

**How to Turn in an Assignment**

Use the following scripts to submit your assignments before the due date as follows:

**PAMT2:** In your cs30x account on the lab workstations or the pi-cluster or ieng6.

```
$ ~/../public/bin/cse30turnin pamt2
```

**Debug2:** In your cs30x account on **pi-cluster**. You will not be able to turn-in debug2 on ieng6 or the lab workstations.

```
$ ~/../public/bin/cse30turnin debug2
```

Always verify your turnins.