BCIT

COMP 8005 Assignment 1

Christopher Eng

A00842832

January 27, 2015

Introduction

The objective of assignment 1 was to compare the performance and efficiency of a program using multiple processes (workerprocs) with one using multi-threading (workerthreads). This was done by having said programs create five processes/threads and complete the same calculation with each. The calculation time was tracked and compared to reach a conclusion. The computation used in this task was a combination of a mathematical calculation using the GMP library that factored a number down to its prime number multiples and a write to a text file. Results are stored in the text files, which are overwritten with each run of the program. The reason behind using a write to a text file as my I/O task is because it is intuitive for a program to do this. This creates a realistic multi-process/-threading scenario. For workerprocs, the text file generated is called workerprocs_output.txt. For workerthreads, the text file generated is called workerthreads_output.txt.

Environment Setup

1. workerprocs

Workerprocs is the program utilizing multiple processes. It is comprised of the following source code files, found in the workerprocs folder of the submission disk:

- workerprocs.c
- primedecompose.c
- primedecompose.h

To compile an executable, the GMP library must be linked to the compiler.

"gcc –Wall –o workerprocs workerprocs.c primedecompose.c –lgmp"

Runtime: "./workerprocs <number to be factored> <optional: number of child processes>"

The default number of processes created is 5, meeting the specifications of the assignment.

2. workerthreads

Workerthreads is the program utilizing multi-threading. It is comprised of the following source code files, found in the workerthreads folder of the submission disk:

- workerthreads.c
- primedecompose.c
- primedecompose.h

To compile an executable, the GMP and pthread libraries must be linked to the compiler.

"gcc –Wall –o workerthreads workerthreads.c primedecompose.c –lgmp –lpthread"

Runtime: “./workerthreads <number to be factored> <optional: number of threads created>”

The default number of threads created is 5, meeting the specifications of the assignment.

Pseudocode

1. workerprocs

Read number of arguments

Set process count based on argument/default

Open file IO

Create result file and write header text

Close file IO

Open pipe

Loop ‘count’ number of times

    Fork new child process

    If child process, run child()

If parent process, run parent()

Child(number to factor, pipe)

Get child process PID

Close pipe read descriptor

Store start time

    Calculate prime number decomposition

    Open file IO

    Write PID and calculated results to file

Store stop time

Calculate time difference between stop time and start time

Write time difference to file

Close file IO

Write time difference to pipe

Return

Parent(# of processes, pipe)

Initialize result to 0

Close pipe write descriptor

Loop ‘count’ number of times

    Read from pipe (blocking)

    Add pipe value to result

Open file IO

Write results to file (total/average)

Close fie IO

Return

2. workerthreads

Create global variables: mutex, character buffer, and data read flag

Read number of arguments

Allocate space for thread function argument

Set thread count based on number of argument/default

Initialize thread function argument to factored number argument

Open file IO

Create result file and write header text

Close file IO

Loop ‘count’ number of times

    Create new pthread, running child()

Run parent()

Free allocated space for thread function argument

Child(number to factor)

Get thread ID

Store start time

    Calculate prime number decomposition

    Open file IO

    Write thread ID and calculated results to file

Store stop time

Calculate time difference between stop time and start time

Local data saved flag false

While data saved is false

    Obtain mutex lock

    Check if data read flag is true

        Set character buffer as time difference

        Write time difference to file

        Set data read flag to false

        Set data saved flag to true

        Close file IO

    Release mutex lock

Return

Parent(# of threads)

Set local count to 0

Initialize result to 0

While count is not # of threads

    Obtain mutex lock

        Check if data read flag is false

        Add character buffer to result

        Set data read flag to true

Increment count

Release mutex lock

Open file IO

Write results to file (total/average)

Close file IO

Return

Testing

1. Does the program create the correct number of processes/threads?

Programs print out whenever a child process or thread is created. By not submitting an argument for number to create, the default is 5. The programs correctly change this amount based on the user submission.

2. Is the prime number decomposition working properly?

Results from the decomposition are printed to the file, so they have been compared to the results of the original source code. Behaviour is as expected.

3. Does the file IO work properly?

The programs initially are responsible for overwriting any existing created file. Then, as the programs run, they write additional information to the file. The formatting is as expected and matches the runtime results of the programs.

4. Is the time calculation working properly?

This functionality was originally tested in by printing the start and end time to the output file. There were no issues with results.

5. Are the final results correct?

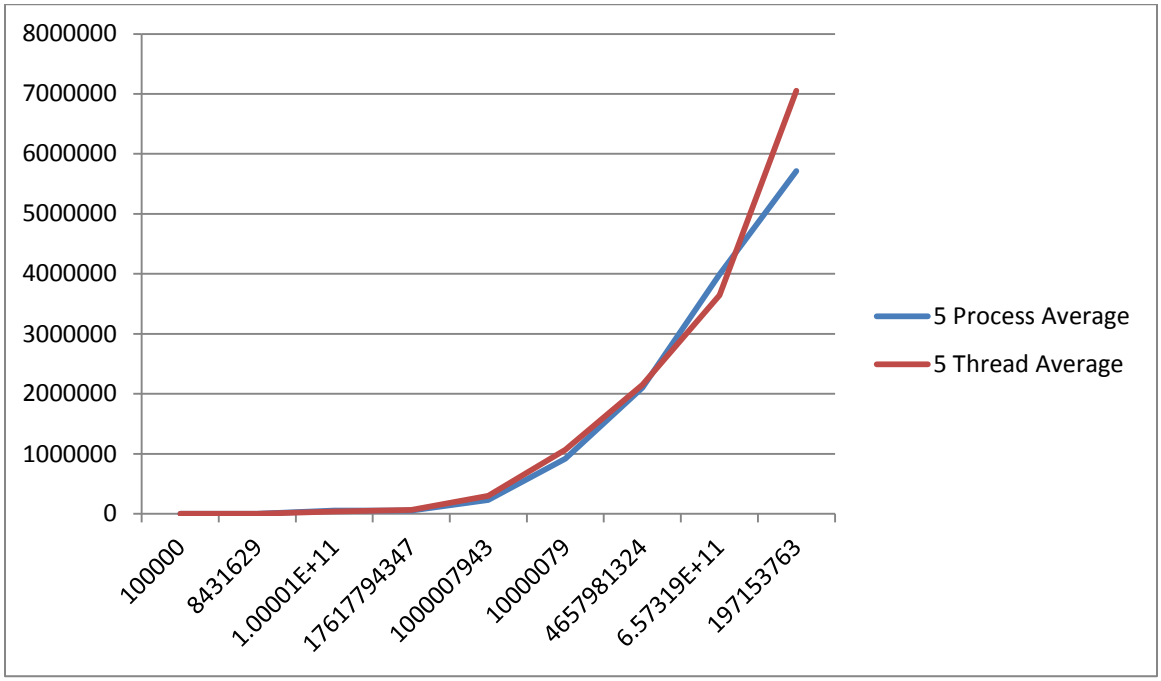
The final results must loop based on the number of child processes or threads. Since it is able to and the sum of the results in the output file match the total, we conclude this to be correct.

Results

Statistics were formed by running each program onto 10 random numbers. This process was done twice, once using 5 processes/threads and another time using 20 processes/threads. The order of runtime for the two programs was alternated with each number, in hopes of creating a fair environment. Each table is ordered by runtime duration for the 5-process test.

5 processes/threads

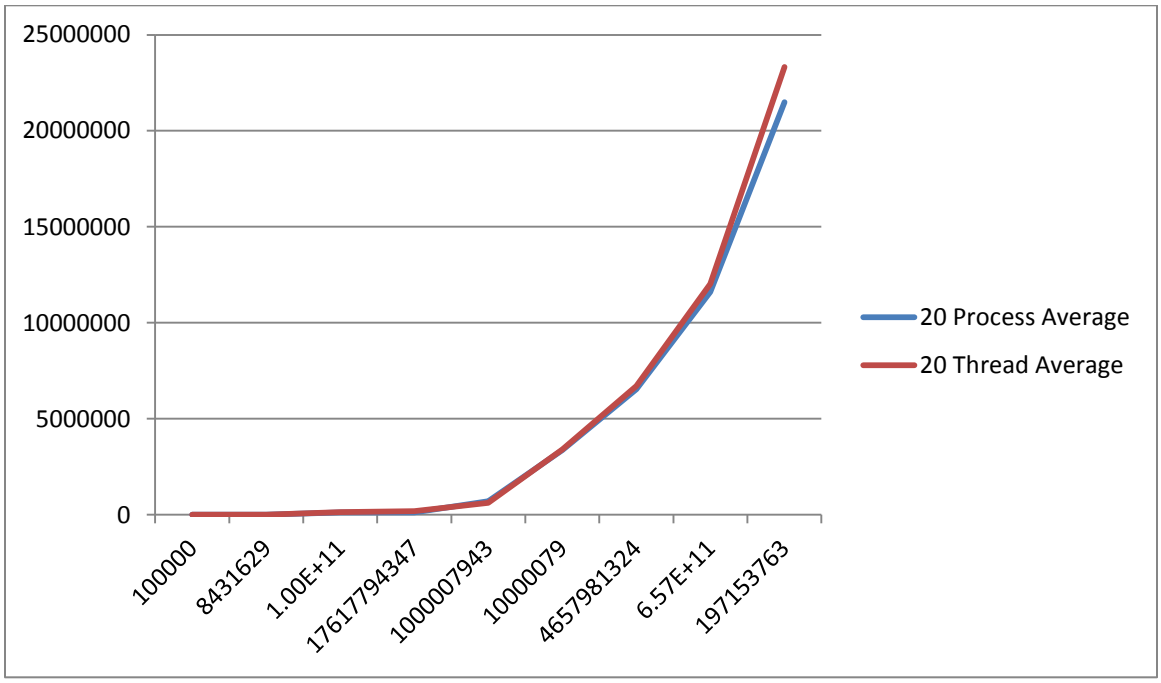
Average (ms) for Number Factored	Process	Thread
100000	162	32
8431629	301	211
100000794345	55182	41722
17617794347	55676	65089
1000007943	232382	298333
10000079	919598	1067976
4657981324	2101800	2151259
657318945648	3993007	3646620
197153763	5716557	7054130



20 processes/threads

Average (ms) for Number Factored	Process	Thread
100000	125	33
8431629	1389	149
100000794345	131485	138016
17617794347	120263	187229
1000007943	705916	615235
10000079	3377394	3405570
4657981324	6536065	6710296
657318945648	11614903	12022023

197153763	21497864	23332729



My analysis was the same for both result sets. The general trend I saw from the runtime data showed that for the shorter calculations, threads were faster than processes. On the data tables, the slower result is shown in red. However, once the calculations began to take longer due to more complex numbers, processes became faster than threads. My initial hypothesis was that threads would be faster than processes, because creating and killing new processes seemed like it would be a more intensive action. The trends seen from the data and charts lead me to believe that starting and closing threads is faster than starting and closing processes. The change in trend for longer calculations brings me to the conclusion that context switching between threads is slower than context switching between processes.

Although this was the conclusion reached, the complexity of the thread program was much higher than that in the process program. This may have caused the thread program to take longer as the runtime continued. The reason behind this complexity was mutex locking to ensure reading and writing to global variables was secure. Although the thread program did not track this portion of the thread function in its timing calculation, the context switching required to keep these threads running even though all computation was finished may have made the results slower than they seemed. A quick review of the stored execution times for each sequential completed process/thread did not show a trend based on runtime order. For example, one would expect the first process/thread to complete fastest, since it would not be sharing calculation time with as many processes and threads immediately. However, on the reverse end, the prime number computation can become faster due to caching.

In conclusion, while I can think of many hypotheses about why the trend I found came to be, the most direct interpretation of the results are that threads are faster at creation and destruction than processes. However, processes seem faster at context switching and multitasking.