High Performance Computing

Homework #6

Due: Wed Nov 3 2021 by 11:59 PM

Email-based help Cutoff: 5:00 PM on Tue, Nov 10 2021

Maximum Points: 20

Submission Instructions

This homework assignment must be turned-in electronically via Canvas. Type in your responses to each question (right after the question in the space provided) in this document. You may use as much space as you need to respond to a given question. Once you have completed the assignment upload:

1. The MS-Word document (duly filled) and saved as a PDF file named with the convention `MUid.pdf` (example: **`raodm.pdf`**)

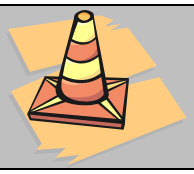
Note that copy-pasting from electronic resources is plagiarism. Consequently, you must suitably paraphrase the material in your own words when answering the following questions.

Name: John Doll

Objective

The objective of this homework is to review the necessary background information about shared memory parallelism using OpenMP.

Read subchapter 7.1 from E-book "Introduction to Parallel Computing" (all students have free access to the electronic book). Links available off Syllabus page on Canvas.



Although the Safari E-books are available to all students there are only a limited number of concurrent licenses to access the books. Consequently, do not procrastinate working on this homework or you may not be able to access the E-books due to other users accessing books.

1. In the space below tabulate three significant differences contrasting implicit and explicit parallelism: [**1 points**]

Implicit Parallelism	***Explicit Parallelism***
Auto or semi-auto realized	Manually realized
Microprocessor and compiler orchestrate parallelism	Programmer orchestrates paralellism
Easily realizable for any program	Requires considerable effort

2. Using Figure 7.1 from the textbook, briefly describe the logical machine model of a thread-based programming paradigm [**1 points**]

Figure 7.1a shows several processes accessing a shared access space and operating on the data concurrently. Figure 7.1b shows the process keeping a local copy of the memory for each thread so that that they aren't all operating on the same data. However, accessing the cache for each of the threads does skew memory access time.

3. In the space below tabulate 2 significant differences between task parallel and data parallel applications: [**1 point**]

Task Parallel	**Data Parallel**
Each thread/process performs different computations	Each thread/process performs the same computation
Data for each thread/process is the same	Data for each thread/process is different

4. The following questions deal with explicit multi-threading: [**2 points**]
   a. What is explicit multi-threading?

   Explicit parallelism requires specification of and interaction between concurrent tasks through communication and synchronization.

   b. State 2 advantages of this approach

   It can maximize efficiency as the programmer has full control of threads and resources

   Programmer can control scheduling and load balancing between threads

   c. State 2 disadvantages of this approach

   It requires rewriting programs to effectively use multiple threads

   The programmer must handle all race conditions

5. Describe (2 to 3 sentences) "Programmer Guided automatic Multi-threading" approach used by OpenMP? How is it different from explicit multi-threading? [**2 points**]

   It is different because we as programmers provide special constructs to identify parts of the program to run with multiple threads and the compiler handles the task of creating, syncing and destroying threads. With explicit multi-threading, the programmer must explicitly program the threads into existence and sync them up and destroy them, whereas OpenMp would handle that for us.

6. Briefly describe one example application (such as: video encoding, image processing, etc.) for data parallel and task parallel applications. Your examples cannot be one of those already mentioned in lecture notes [**1 point**]
   a. Example of a data parallel application

   A data parallel application could be adding all the numbers from 1 to 1000 together. Each thread would be performing the same computations, but with a different section of numbers. Four threads could separate the 1000 numbers into 4 equal sections and report back their sums as they completed.

   b. Example of a task parallel application

   An example of a task parallel application would be check to see how many times a single text file had a specific pattern. So thread A would check for how many

times pattern A is matched, and thread B would check for how many times pattern B is matched, and so on.

7. Consider the following OpenMP program that is being run using 4 threads

```
#include <iostream>
#include <omp.h>

int main() {
  #pragma omp parallel
  { // start parallel
    int numThreads = omp_get_num_threads();
    int threadID   = omp_get_thread_num();
    std::cout << "hello OpenMP from " << threadID   << " of "
              << numThreads << " threads.\n";
  } // end parallel
  return 0;
}
```

Actual output from above program (with 4 threads) copy-pasted below:

```
hello OpenMP from hello OpenMP from hello OpenMP from 0 of 4 threads.
3 of 4 threads.
1 of 4 threads.
hello OpenMP from 2 of 4 threads.
```

a. Why does the output appear garbled as shown above? [**1 point**]

The threads are all sent off at the same time and instead of making sure they complete sequentially, they complete at their own pace, meaning they print to std::cout as soon as they finish, not in order.

8. What is the output from the adjacent OpenMP program when it is compiled and run as shown below? Briefly describe how you determined the output from the program? [**2 points**]

```
$ g++ -fopenmp q8.cpp -o q8
$ export OMP_NUM_THREADS=4
$ ./q8
```

```
#include <iostream>
#include <omp.h>

int main(int argc, char *argv[]) {
#pragma omp parallel if (argc > 1)
  { // fork
    std::cout << "Output #1\n";
  } // join

#pragma omp parallel num_threads(1)
  { // fork
    std::cout << "Output #2\n";
  } // join
  return 0;
}
```

Output #2
The first output only happens if an argument parameter is specified when running the program, and since no arguments are passed, argc is just 1 for the path of the file being executed.

9. What is a race condition? How does a race condition impact outputs from a program or results from a method? [**1 point**]

A race condition is when multiple threads update the same variable concurrently. It changes the output results from a method because thread A can give the variable a value, but once thread B completes, it will give the variable the value that it has computed, overwriting what thread A just assigned to it.

10. Parallelize the following data parallel method using OpenMP using a parallel block by manually computing starting and ending index/bounds for each thread (**no**, you cannot use `omp parallel for construct`) [**3 points**]

```
// Assume this method is correctly implemented
bool isPrime(int num);

std::vector<bool> isPrime(const std::vector<int> numList) {
std::vector<bool> primes(numList.size());
#pragma omp parallel
   {
       const int iterPerThr = numList.size();
       const int startIndex = omp_get_thread_num() * iterPerThr;
       const int endIndex = startIndex + iterPerThr;
      for (size_t idx = startIndex; (idx < endIndex); idx++) {
            primes[idx] = isPrime(numList[idx]);
       }
   }
   return primes;
}
```

11. Exploring "Programmer Guided automatic Multi-threading" via parallel algorithms in C++ [**2 points data + 3 points for inferences = 5 points**]

Background: The C++ Standard Library implementation provided with the GNU C++ compiler called `libstdc++`) also provides a "parallel mode". Using this mode enables existing serial code to take advantage of many parallelized standard algorithms that enables effective use of multi-core processors or multi-CPU machines. The parallel mode operations can be enabled using compiler flags (namely: `-fopenmp -D_GLIBCXX_PARALLEL`) to enable use of parallel mode of operation whenever possible. The compiler and standard library use heuristics to decide if an algorithm should be run in parallel. Consequently, some of the algorithms may not be run in parallel mode for your specific program/data. Note that this mode of operation does not require any changes to the program – however this is still explicit parallelism – just that some other programmer has written the library for us using OpenMP.

For this part of the exercise, you are supplied with a slightly modified version of the solution from a prior exercise in this course in `/fs/ess/PMIU0184/cse443/homeworks/homework6`.
Do NOT modify the program but study it. You just have to run it (on `pitzer.osc.edu`) using the supplied SLURM script.

Once the job completes, collect runtime statistics from the output file in the following table:

Threads	Run #	User time	System time	Elapsed time	%CPU
1	1	52.65	0	52.78	99
1	2	49.48	0	49.61	99
1	3	51.95	0	52.09	99
1	4	53.59	0	53.73	99
1	5	53.83	0	53.97	99
1	**Average**	**52.30**	**0**	**52.436**	**99**
2	1	54.86	0	27.67	198
2	2	53.90	0	27.15	198
2	3	54	0	27.31	198
2	4	53.82	0	27.14	198
2	5	53.87	0	27.11	198
2	**Average**	**54.09**	**0**	**27.27**	**198**
4	1	53.55	0	15.06	355
4	2	53.52	0	15.03	355
4	3	53.3	0	14.95	356
4	4	53.29	0	14.98	355
4	5	53.2	0	14.95	355
4	**Average**	**53.37**	**0**	**14.99**	**355**
6	1	55.66	0	9.7	573
6	2	55.6	0	9.69	573

6	3	55.57	0	9.71	572
6	4	56.22	0	9.77	575
6	5	56.55	0	9.84	574
6	**Average**	**55.92**	**0**	**9.74**	**573**
8	1	58.59	0	7.86	744
8	2	58.61	0	7.86	745
8	3	57.89	0	7.77	744
8	4	58.47	0	7.86	743
8	5	58.28	0	7.84	742
8	**Average**	**58.37**	**0**	**7.83**	**744**

Speedup S is defined as: S = Ts ÷ Tp, where Ts is serial runtime (recorded in Task 2) and Tp is parallel runtime. Note that in theory, under ideal conditions, given *n* threads it is desirable to attain a speedup of *n*. Consequently, the theoretical runtime with *n* threads is computed using the formula T_{THEORY} = Ts ÷ *n*. Using these relationships (or formulas) compute the expected theoretical and observed speedups using a varying number of threads:

Serial runtime	Threads	Theoretical or expected runtime	Observed Runtime	Observed Speedup
53.75	1	53.75	**52.436**	1
53.75	2	26.875	27.27	1.97
53.75	4	13.43	14.99	3.58
53.75	6	8.95	9.74	5.51
53.75	8	6.72	7.83	6.86

Briefly (3 to 4 sentences each) describe your inferences from the experiment in the space below:

1. Does the user time vary a lot when using multiple threads? Why not?
Not really, because we are still doing about the same number of operations. User time measures how much work the hardware is doing and how much time's worth of work is being done. If we do two threads, it measures how long each thread does work, and that makes up the user time. When we run a program, if it is written correctly, adding threads should not alter user time too much.

2. Does the elapsed time vary a lot? Why?
Yes, because the elapsed time is the wall clock time. If we do 4 threads, then it is like 4 people doing a job instead of one person, and since 4 people (in general) can work faster than 4, we see that reflected In our elapsed time. Now, it isn't quite the magnitude of the number of threads, but it is relatively close, and we can tell that more threads decreases the elapsed time.

3. Does the system time vary with threads? Why?
No, the system time does not vary because no system calls are being made.

4. Does the %CPU increase with number of threads? Why?
Yes, because the program measures how many CPU's are being used on average when running the program. If one thread is being used, max CPU usage would be 100%, if two threads were being used, max CPU usage would be 200% and so on. %CPU takes into account how many threads are doing work and how efficiently they are using their cores.

5. Does the program achieve theoretical speedup? Why not?
Not quite. Theoretical speedup would be n number of threads, but we don't quite hit n. We are close early on, but differ more and more as threads are added. This is due to overhead in creating threads, syncing threads, and removing threads when they are no longer needed.

6. Although not recorded in tables above, you should notice a slight growth in memory usage of the program when number of threads are increased. Why?
The threads will have to have their own local storage. In order to be parallel without creating race conditions while updating shared values, we need to have threads work on their own variables and report those back as they finish so we can process each of them individually. Otherwise, the threads would be writing over the shared variables.