Basics and Maths in OpenMP

January 16, 2022

Contents

1 Exercise 1 **2**
1.1 Solution . . . 2

2 Exercise 2 **3**
2.1 Solution . . . 3
2.1.1 Static Scheduling . . . 4
2.1.2 Dynamic Scheduling . . . 4

3 Exercise 3 **5**
3.1 Solution . . . 5
3.1.1 Benchmark . . . 5

1 Exercise 1

Write a program in C (basics.c) that implements parallel programming with OpenMP and calling the appropriate functions will display on the screen:

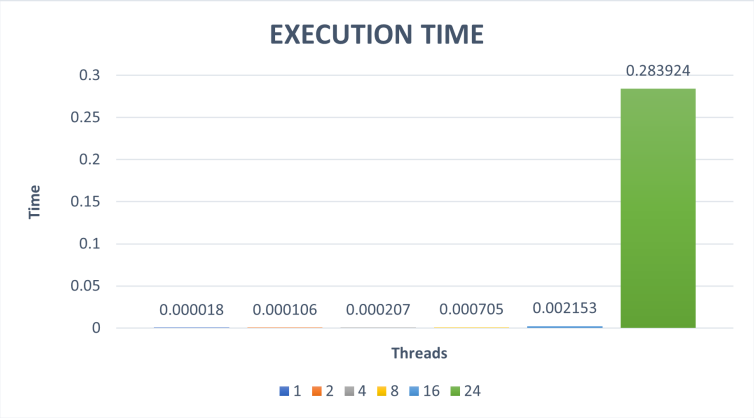
1. how many processors the computer running the program has.
2. maximum number of computer threads.
3. number of threads participating in the execution.
4. message from the master thread: "Hello. I am the master thread."
5. message from the other threads: "Hi there! I am thread *", where * = their id
6. the time of execution.
7. Add a table inside your report, which will show the number of threads and the corresponding execution time on your computer.

Hint: Avoid id-based controls and use the appropriate functions to vary execution by thread.

1.1 Solution

Function calls used for each sub-problem.

1. omp_get_num_procs(): Returns the total number of logical processors.
2. omp_get_max_threads(): Returns the maximum number of available threads.
3. omp_get_num_threads(): Returns the number of threads participating in the execution.
4. omp_get_thread_num(): Return the thread id.
   - Thread with id = 0 is the master thread
   - Thread with id other than 0 is the master thread
5. A clock is used to calculate the time of execution.
6. The relation between threads and execution time can be seen below.



The overhead when more threads are spawned can be attributed to the fact that spawning a thread is a complicated process that takes a lot of time. Since our CPU only has 24 threads and each thread is spawned only once, the spawn overhead is not actually a huge limiting factor. The operation that is truly responsible and slows the execution even further is that each thread performs an I/O task (printing to screen with printf) which are infamous for their speed since they require at least one system call. The more threads we spawn the more I/O tasks are performed and thus the slower the execution gets.

2 Exercise 2

Write a program in C (pi.c) that implements parallel programming with OpenMP and calculates an approximation of pi, using the formula

$$pi = \int_0^1 \frac{4}{1+x^2}\, dx$$

Use Simpson's rule to approach the integral, i.e.,
if $x_i = i * dx$, i=0,1,...,N, $dx = \frac{1}{N}$, a partition of [0,1] with N intervals (N + 1 points)

$$pi = \int_0^1 \frac{4}{1+x^2}\, dx \approx \frac{dx}{3}(y + 4y_1 + 2y_2 + ... + 2y_{N-2} + 4y_{N-1} + y_N)$$

where $y_i = \frac{4}{1+x_i^2}$, i = 0,1,2, ..., N.

Your program should:

1. run repeatedly in terms of the number of threads, from 1, 2, 4, 8, 16 and 32. In each iteration.
2. The sum will be broken into a partial sums and each thread will calculate a partial sum. Try chunksize = 10, 100, 1000.
3. The assignment of the terms of the sum to the threads will be static / dynamic (schedule=static/dynamic). Avoid checks based on which repetition is performed to have the appropriate factor in the sum as this will cause unnecessary delays.
4. Measure the execution time in each repetition and the number of operations per sec for all chunksizes and scheduling type. The message should appear on the screen: pi is approximately * , computations time = *, number of threads = *, FLOPS = *, chunk = * , scheduling = *
5. Add a table inside your report, which will show the number of threads and the corresponding execution time on your computer for all chunksizes and scheduling type. Comment on the results. Are the times similar or different and why?

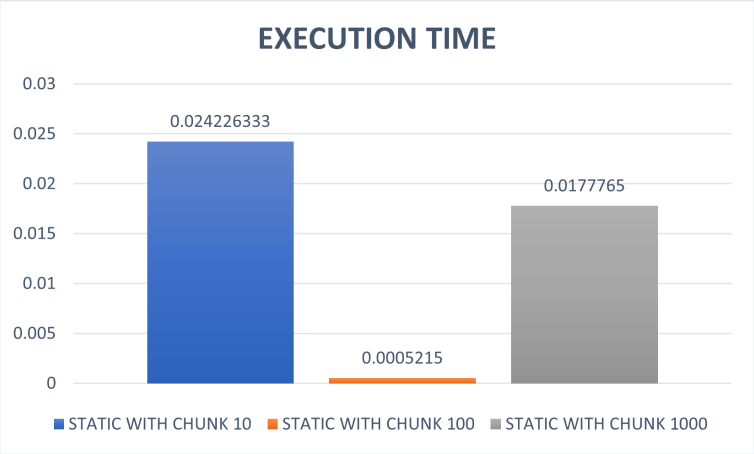
Hint: FLOPS = number of operations per sec. Use $N = 10^8$.

2.1 Solution

Each thread takes a chunk of the total iterations. To completely get rid of critical sections each threads accesses it's own memory space. After all threads have finished the master thread accumulates the computed values of each thread and prints the final result. We experimented with different chunk sizes, number of threads and scheduling policies and the results can be seen in the benchmark below.

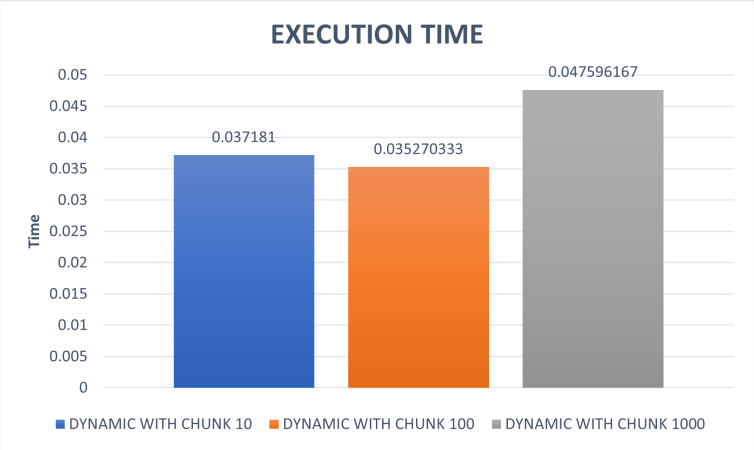
2.1.1 Static Scheduling

As it can be observed below larger chunk sizes make the execution faster when less threads are available. This can be attributed to the fact that fewer threads are spawned. The actual computation does not take that much time and thus the overhead of spawning threads seems larger compared to the actual computation.



2.1.2 Dynamic Scheduling

Dynamic scheduling is slower than static. Each thread has a certain number of operations to perform and then searches for the next chunk of data that was not already processed. Since all threads roughly perform the same operations they all finish at the same time. Each thread now tries to find the next available chunk which produces an unnecessary overhead. If the number operations that each thread performs where totally different then the dynamic scheduling would probably outperform static scheduling.



3 Exercise 3

Write a program in C (jacobi.c) using OpenMP, which implements the Jacobi method for system solution where N is the dimension of the system

$$Ax = b \Leftrightarrow \begin{bmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & -1 \\ & -1 & 2 & \end{bmatrix} x = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ N+1 \end{bmatrix}$$

1. runs for M iterations
2. in each iteration it should calculate the new approach of the solution, xnew, (and display on the screen) the remainder of the approach, i.e. || b - A xnew || 2 , and the difference of the last 2 approaches, i.e. || xold - xnew || 2 , with appropriate comments, e.g.: iter = *, residual = *, difference = *
3. also display the execution time.
4. run your program for different number of threads (1, 2, 4, 8, 16, 32) and add a table to your report, which will show the number of threads and the corresponding execution time on your computer.

3.1 Solution

The helper matrices that we used can be seen below.

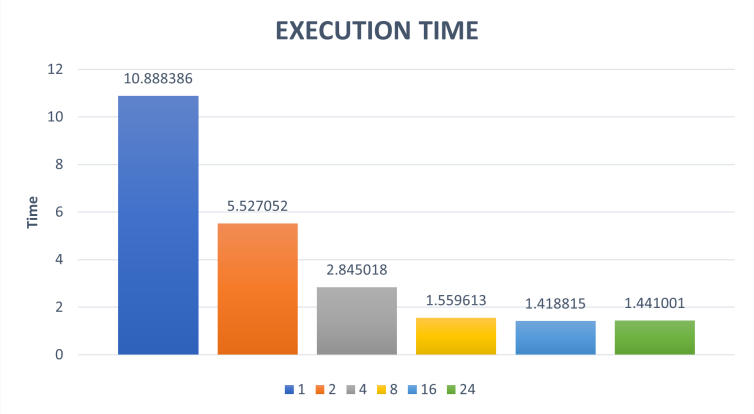
D_inverted

1/2	0	0	0
0	1/2	0	0
0	0	1/2	0
0	0	0	1/2

L_plus_U

0	1	0	0
1	0	1	0
0	1	0	1
0	0	1	0

3.1.1 Benchmark



Since the execution takes a lot more time than the previous exercises the overhead of spawning a thread is a lot less relative to the actual computation. Here more threads allow for better optimization since more tasks can be performed in parallel.