Christopher Pac

CSCI-GA.3033-009

Multicore Processors: Architecture & Programming

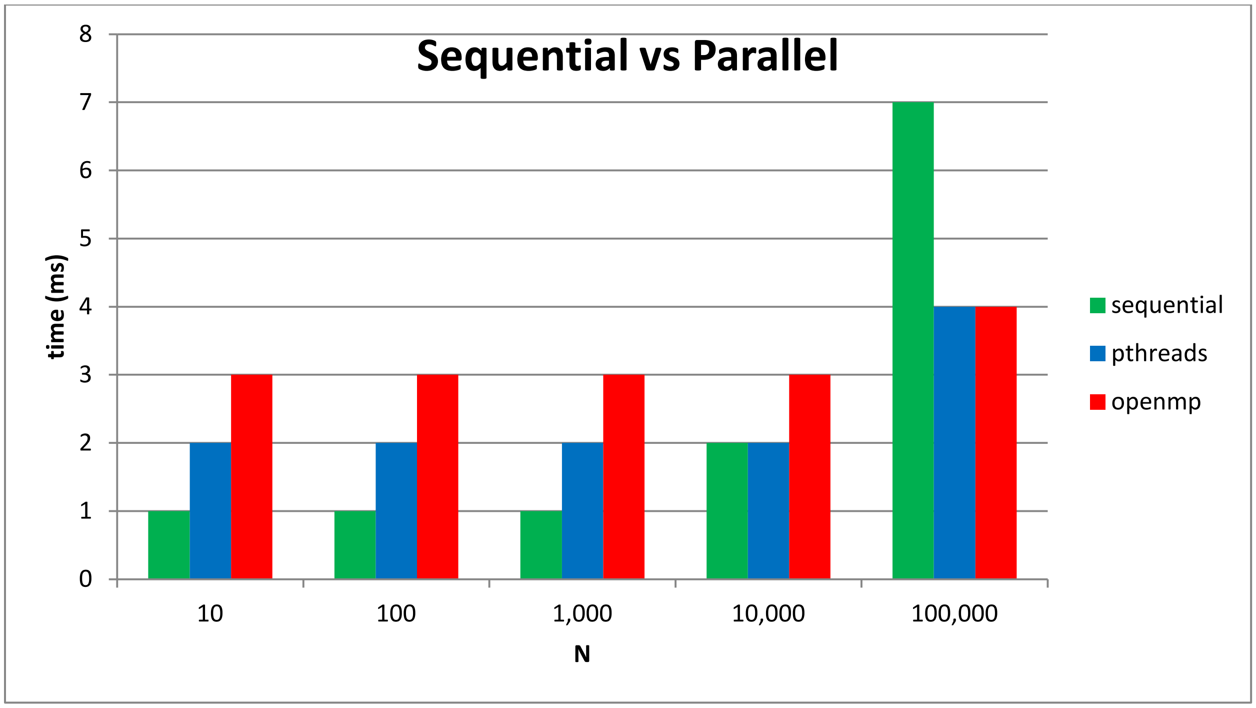
Lab Assignment #2

Section 1 – Compilation

gcc version 4.8.2

Sequential version	==> gcc -o slab mlab2.c
Pthreads version	==> gcc -o plab mlab2pthread.c -lpthread
OpenMP version	==> gcc -fopenmp -o olab mlab2openmp.c

Section 2 – Graph



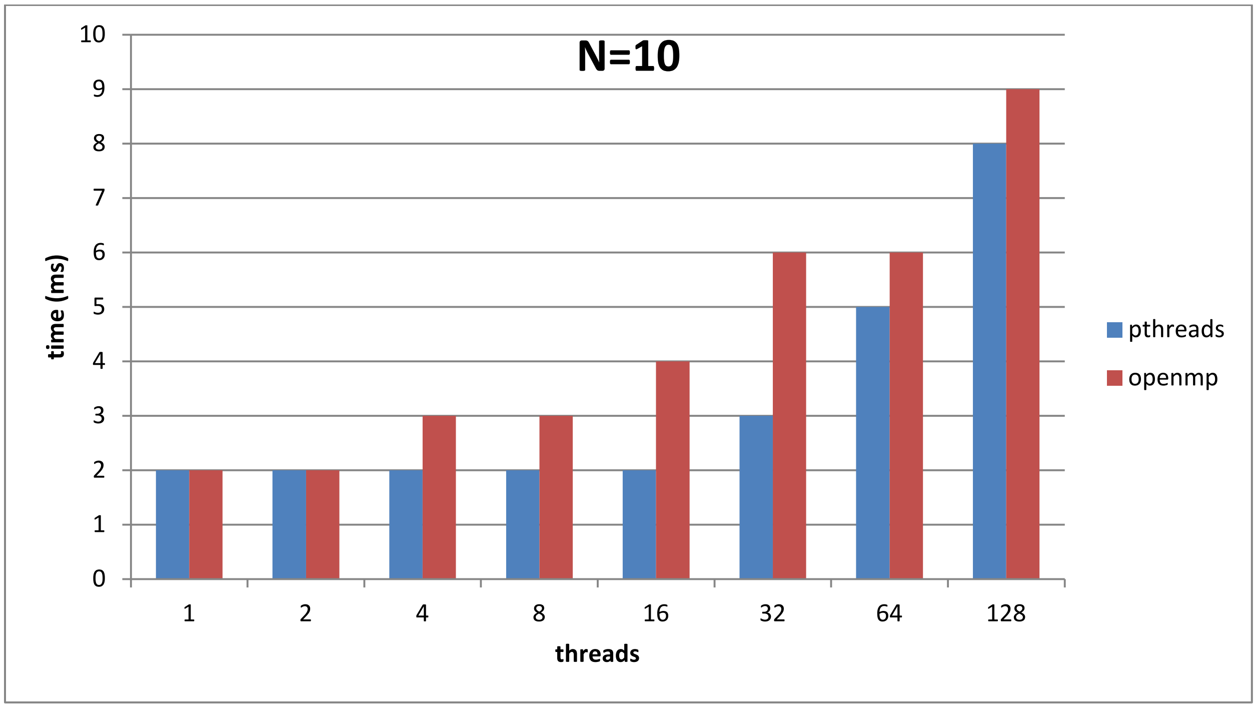
Section 3 – Analysis

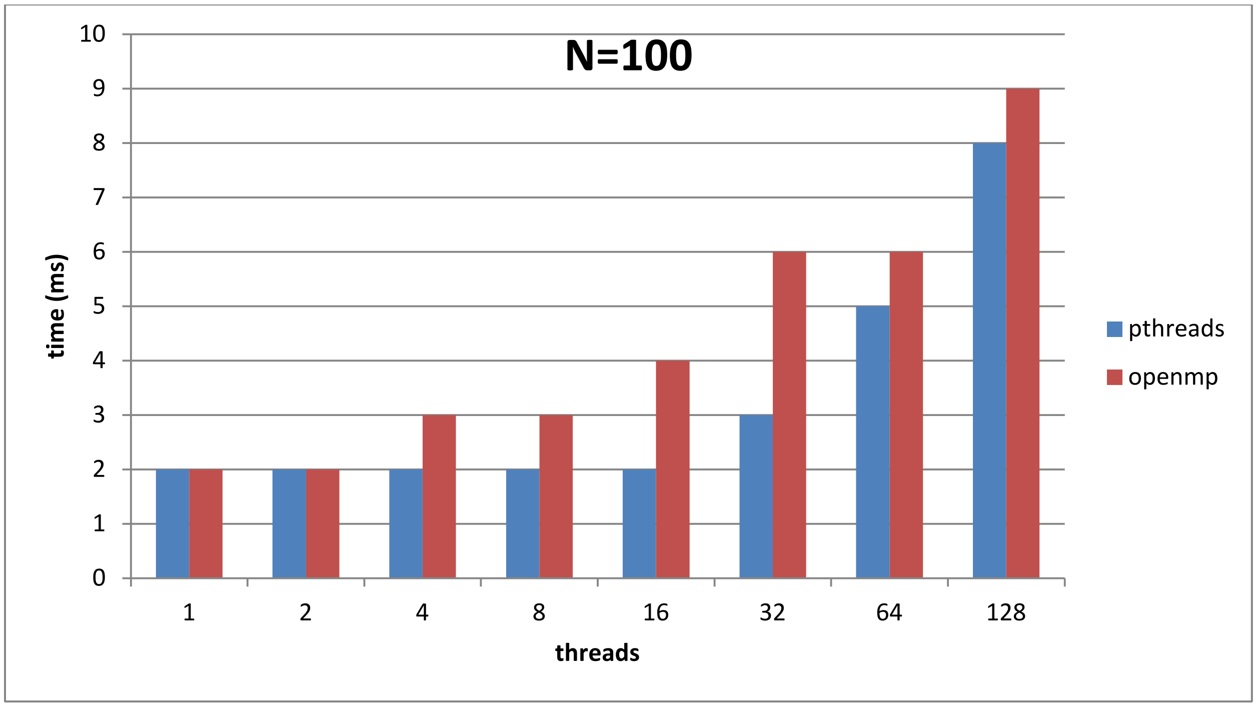
The obvious conclusion from the graph above is that there is a work size cut-off point between sequential and parallel execution time. The cut-off point is the task that is composed of approximately 10k loop iterations. At this point the sequential version performs about as well as the parallel version. At 100k there is enough parallel work for the threads such that parallelizing the work and paying the overhead cost still leads to faster execution than sequential version.

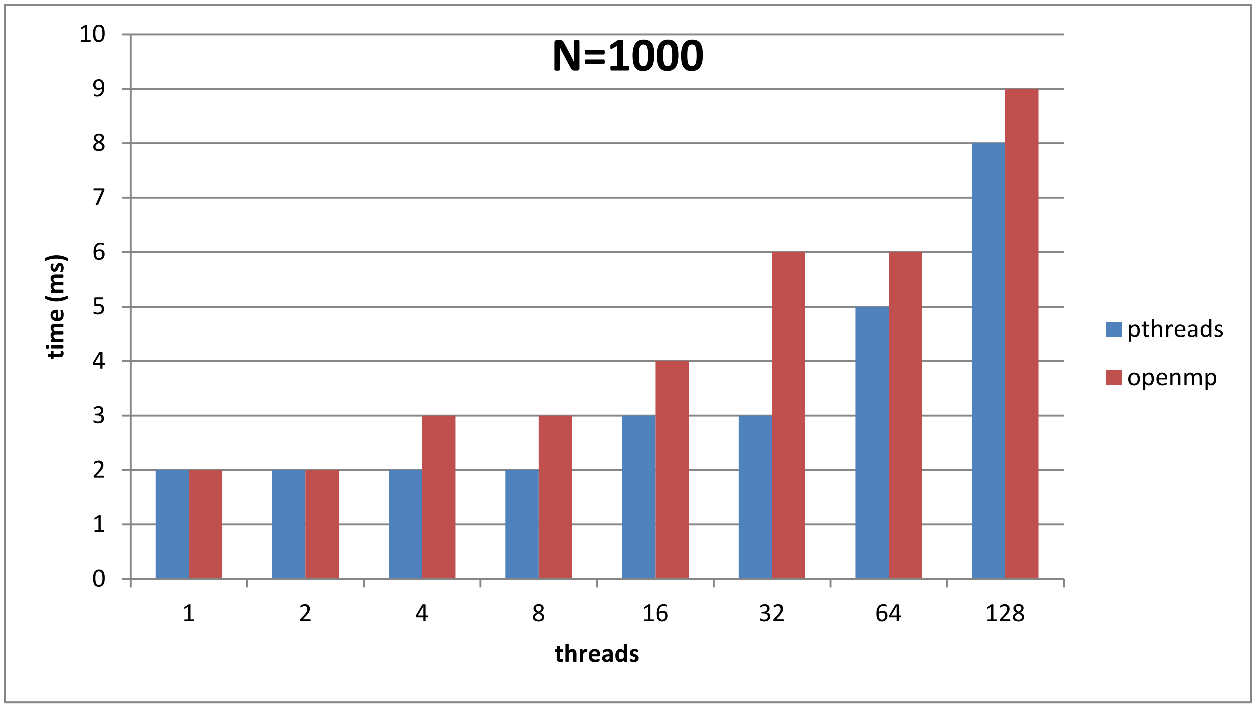
For small task sizes (n=10, 100, 1000) the overhead for managing threads and distributing work is significant part of the total execution time. Based on the graph this overhead is roughly 1 ms, which doubles the execution time for small task sizes. It is clear that for small task sizes there just isn't enough work compared to the cost of parallelization overhead. Furthermore, it appears the overhead for OpenMP is greater than for Pthreads.

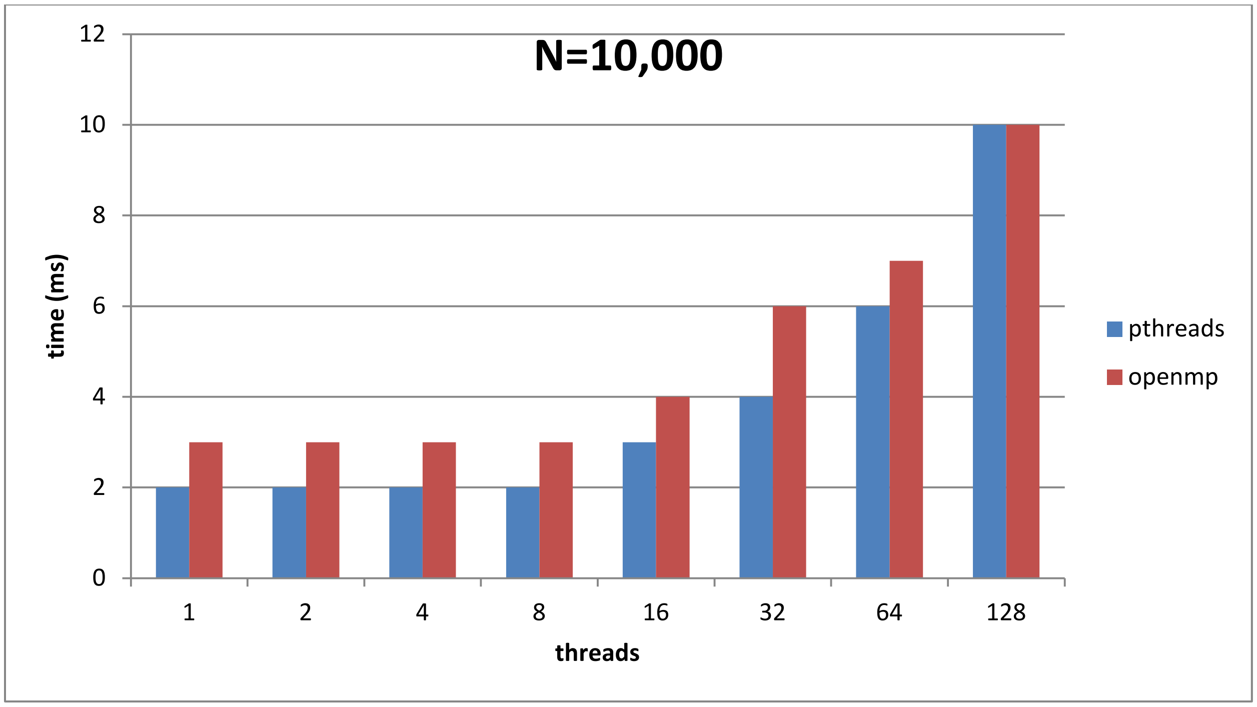
Parallelization is not free and we should always ask the question if we have enough parallel work to justify parallelizing our code. Small tasks are better left on a single core thus saving ourselves the overhead of thread management, work distribution, thread communication, synchronization, etc, and potential parallel bugs.

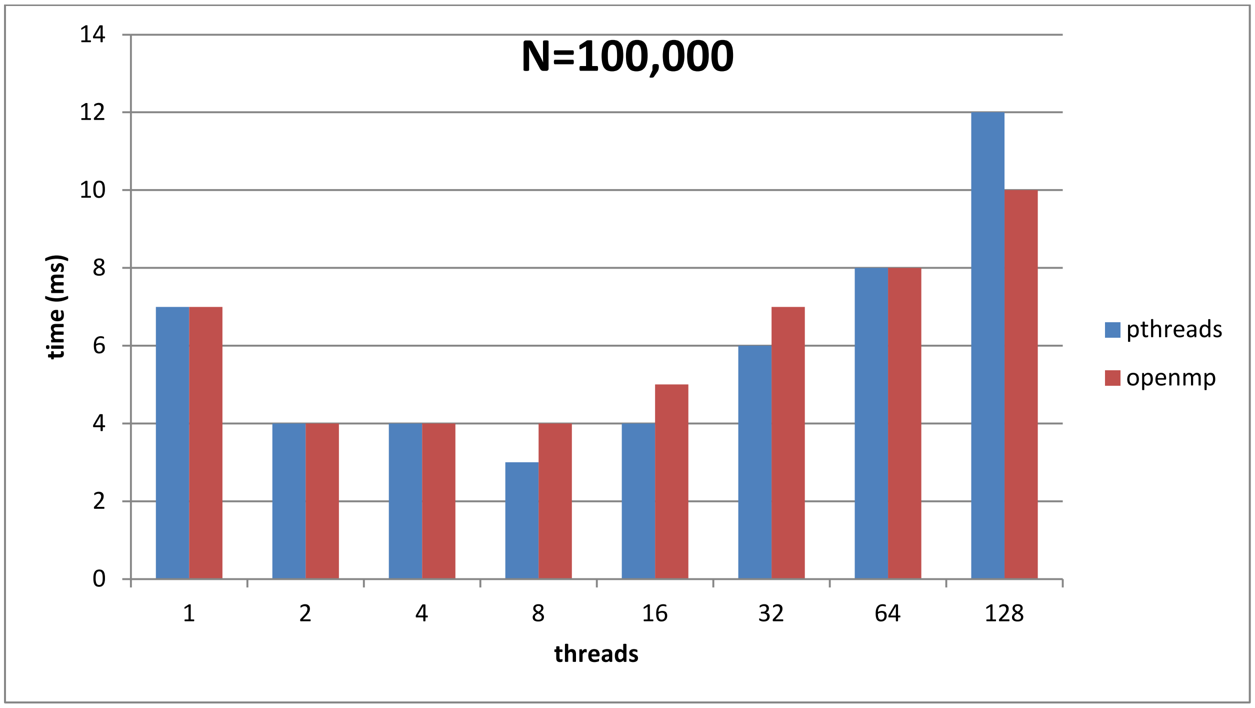
Section 4 – Graphs

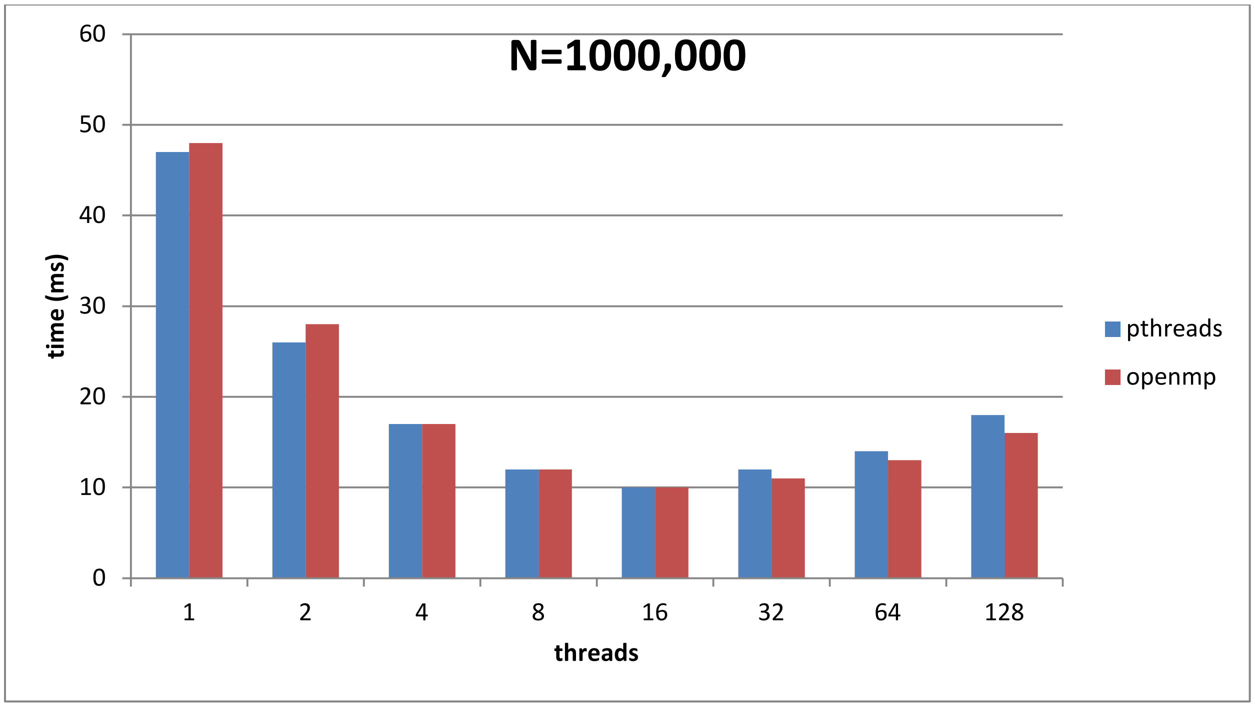



N=100
time (ms)
10
9
8
7
6
5
4
3
2
1
0
1
2
4
8
16
32
64
128
threads
pthreads
openmp

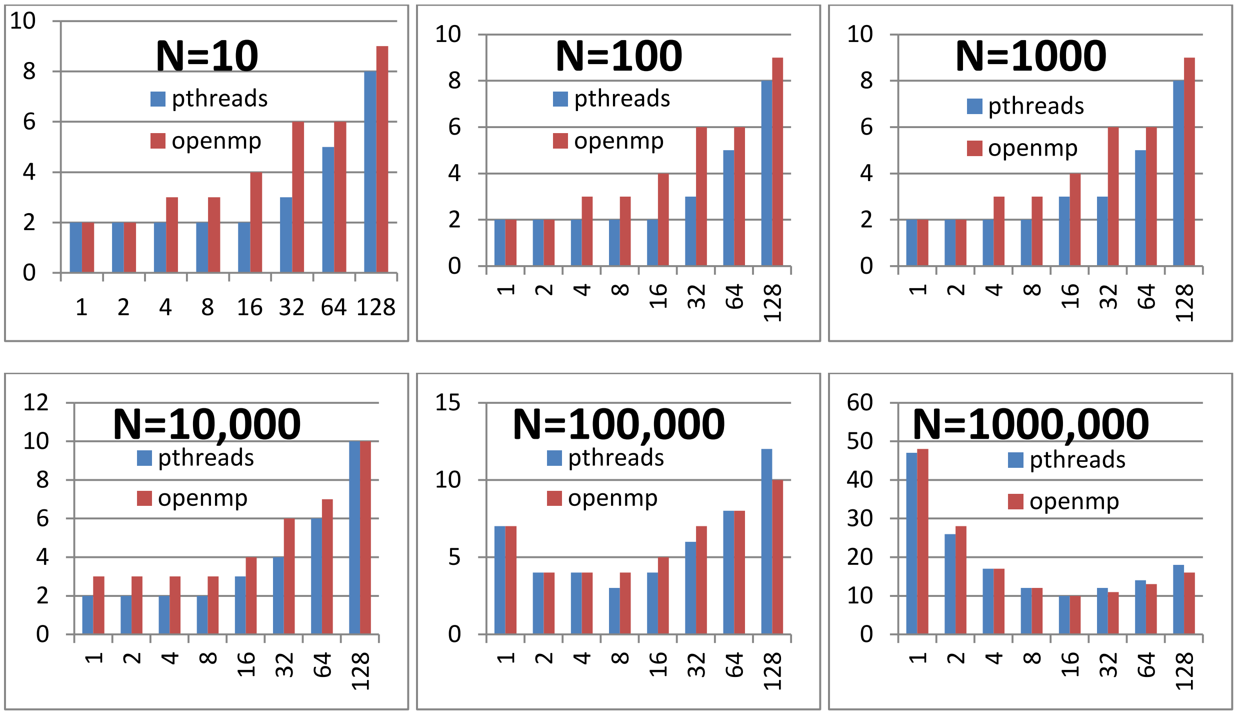

N=1000
time (ms)
10
9
8
7
6
5
4
3
2
1
0
1
2
4
8
16
32
64
128
threads
pthreads
openmp


N=10,000
time (ms)
threads
pthreads
openmp
0
2
4
6
8
10
12
1
2
4
8
16
32
64
128


N=100,000
time (ms)
threads
pthreads
openmp
0
2
4
6
8
10
12
14
1
2
4
8
16
32
64
128



Graph Summary Overview



Section 5 – Analysis

Implementation and Measurement Notes

Both OpenMP and Pthreads are implemented such that the specified number of threads (x-axis) is always created regardless if there is enough work for those threads. For example, in the graph N=10 there are only 10 loop iterations but I still create 1, 2 ,4, 8, 16, 32, 64, and 128 threads. Obviously in this case only ten threads will actually do any meaningful work and the rest will be purely an overhead. This was done in order to measure the impact of unused threads. Another important point that needs to be made is that the measurements have at best ±1 ms precision thus making comparisons based on the graphs between OpenMP and Pthreads somewhat speculative. Furthermore, since the experiment was performed on a shared machine (crunchy3), I have run each program at least twenty times to obtain a median execution time. Lastly, I have noticed performance differences of OpenMP between the two versions of gcc (4.4.7 and 4.8.2) that amounted to about ±1 ms (the graphs are based on gcc 4.8.2).

Graph Interpretations

First the graphs can be broken into two groups:

1) Not enough parallel work: N=10, N=100, N=1000, N=10,000
2) Enough parallel work: N=100,000, N=1000,000

In the first group, the execution time on 1 thread is about the same as on 2, 4, and 8 threads. Since, the execution time necessary to perform the specified work (i.e. N) is about 1 ms, any additional time is purely due to the overhead of managing threads (spawning/tearing, additional memory allocations, synchronization, etc) and any work distribution calculations. So, as the number of threads increases (> 8) the execution time increases because this overhead becomes more pronounced. Also, Pthreads implementation appears to tolerate additional threads much better than OpenMP. This might be due to the fact that Pthreads is a low-level API as opposed to OpenMP. Lower level APIs usually perform faster at the expense of more complexity. Furthermore, in the first group we can observe that the execution time with 8 threads or less is the same. The work is so small that most likely each thread performs its work on the same core and multiple threads have no positive contribution.

In the second group, we can see that increasing number of threads, to a point, decreases the execution time. This point indicates the best work load balance for the number of created threads and after this point the execution time once again increases. After this point the additional threads do not have enough work to perform to justify their creation and their management significantly contributes to the total execution time. For N=100,000 the best load balance is achieved with 8 threads. For N=1000,000 (not required in the lab) the work becomes the most significant part of the overall execution and the best load balance is achieved with 16 threads. However, unlike N=100,000, the additional thread overhead past the optimal load balance point is less significant compared to the total work.

This lab involved “embarrassingly” parallel work with the main work being a *for* loop with relatively small number if iterations. Therefore, there is no surprise that both Pthreads and OpenMP had very similar execution times.