|  |  |  |  |
| --- | --- | --- | --- |
| 3.2.1) **Threads** | **OpenMP for** | **OpenMP task** | **pThreads** |
| 1 | 104.1 | 112.8 | 123.7 |
| 2 | 204.9 | 248.1 | 246.8 |
| 3 | 309.7 | 375.2 | 371.4 |
| 4 | 418.3 | 501.3 | 495.4 |
| 5 | 529.4 | 612.2 | 618.1 |
| 6 | 627.1 | 643.9 | 742.6 |
| 7 | 691.5 | 770.6 | 873.4 |
| 8 | 781.3 | 926.2 | 1001.5 |
| 9 | 662.1 | 1038 | 767.2 |
| 10 | 744.9 | 1054.8 | 855.6 |
| 11 | 792.3 | 1081.7 | 934.6 |
| 12 | 837.6 | 1095.9 | 1027 |
| 13 | 852.3 | 1121.7 | 946 |
| 14 | 871.2 | 1131.1 | 1063.3 |
| 15 | 922.6 | 1151.7 | 1075.5 |
| 16 | 948.4 | 1170.6 | 1143.6 |

3.2.2) The fastest speed achieved was 1170.6 Mflops/sec with OpenMP-task, which is almost 10x faster than the serial computation. Linear scaling was not achieved as we would need to have 16x speed up (about ~1800 Mflops/sec) to call it linear for 16 threads. After 8 threads, the scaling dropped. The overhead cost to handle more than 8 threads outweighs adding more processors. After 8 threads, there is always a dip in performance as the overhead to handle the new processor costs more resources than the resources obtained with an extra tread. The same occurs after 12 processors.

3.2.3) My matrix multiply this about 76x slower compared to that peak. There are many tools such as the Intel intrinsics SSE that I have not used. The matrix multiply method used is also the naïve method. There are other faster techniques that can be used to harness the cache. The proper looping technique to increase the probability of having the value in a lower-level cache drastically increase the speed as well.

3.2.4) I like both for different reasons. OpenMP I am familiar with and the wrapper makes it clear what regions are parallel and what attributes the parallelization has. However, Pthreads keeps the simplicity of forking and joining.

3.26)