Mincheol Sung.

3.2.1

|  |  |  |  |
| --- | --- | --- | --- |
| # of threads | Omp\_for | Omp\_task | Pthreads |
| 1 | 168.637037 | 166.011578 | 166.078513 |
| 2 | 324.048118 | 330.798664 | 323.306110 |
| 3 | 476.936904 | 480.057462 | 479.113188 |
| 4 | 628.436369 | 651.380216 | 629.817026 |
| 5 | 759.639388 | 767.942578 | 754.374734 |
| 6 | 905.212143 | 926.173051 | 892.802645 |
| 7 | 1048.579712 | 1053.718020 | 1044.610255 |
| 8 | 1200.145331 | 1220.849105 | 1202.753864 |
| 9 | 917.885547 | 1203.069713 | 873.176287 |
| 10 | 1013.479569 | 1217.955004 | 985.371492 |
| 11 | 1108.458466 | 1244.481767 | 1088.345654 |
| 12 | 1198.226304 | 1229.988668 | 1189.917319 |
| 13 | 1132.850568 | 1243.831777 | 1057.217254 |
| 14 | 1225.549578 | 1244.365306 | 1165.529612 |
| 15 | 1223.433849 | 1263.954122 | 1165.842191 |
| 16 | 1252.843837 | 1274.633079 | 1239.533094 |

3.2.2

In the case of OpenMP for with 16 threads, the fastest implementation is 7.549534 time faster than the scalar baseline.

In the case of OpenMP task with 16 threads, the fastest implementation is

7.680834 time faster than the scalar baseline.

In the case of Pthreads with 16 threads, the fastest implementation is 7.469324 time faster than the scalar baseline. I did all implementation on the hive machine.

The speed increased until 8 threads, but after that, it dropped down and pop up. This is because the hive machine has 8 cores, so after 8 threads there would be extra overhead to arrange each threads to each cores.

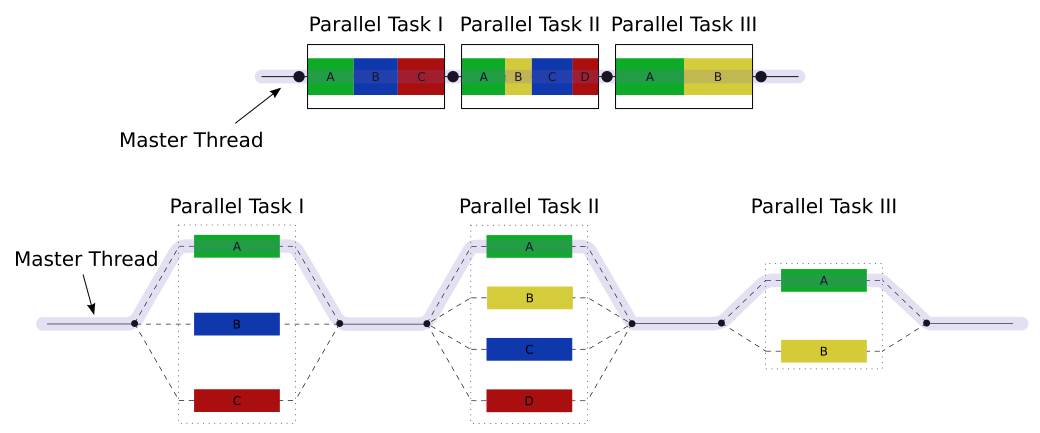
3.2.3

My peak performance of matrix-multiply is 1274 mflops/s with OpenMP\_task. Compared to the hive machine’s peak performance (76.8 gflops/s) it is very slow performance. Using cache, SSE or other optimizing techniques may cause the peak performance of the machine. However, my performance is without that kind of optimization or techniques.

3.2.4

I prefer OpenMP because it is much easier to use that pThreads.

For using OpenMP, I just put the #pragma. However, to use pThreads, I have to know about several pThreads functions.

3.2.5 and 3.2.6

A master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. Like pThreads, after computing, the threads join in.

Actually, I compiled as “g++ -O3 –fopenmp –fdump-tree-ssa –c matmul.cpp”. I got a matmul.cpp.017t.ssa file, but I couldn’t understand what there is in the file.

4.1.1 and 4.1.2

|  |  |  |
| --- | --- | --- |
| # of threads | -n1 | -n10 |
| 1 | 0.00795197(sec) | 0.0551 (sec) |
| 2 | 0.00353503 | 0.0287561 |
| 3 | 0.00238204 | 0.0208681 |
| 4 | 0.00183606 | 0.015645 |
| 5 | 0.00158501 | 0.01126 |
| 6 | 0.00130296 | 0.0118451 |
| 7 | 0.00118113 | 0.00805116 |
| 8 | 0.00173283 | 0.0108092 |
| 9 | 0.00162005 | 0.015373 |
| 10 | 0.00146008 | 0.0110838 |
| 11 | 0.00135684 | 0.010293 |
| 12 | 0.00127006 | 0.0102541 |
| 13 | 0.00129294 | 0.00964308 |
| 14 | 0.00120997 | 0.00894117 |
| 15 | 0.00125098 | 0.00781918 |
| 16 | 0.00119305 | 0.00711489 |

In “–n1 computation”, it seems like the implementation goes down with more threads. However, after the 8 threads, the implementation time goes up a little bit, because of the number of the cores. In “-n10 computation”, the speed varies lot. Before 8 threads, the implementation time goes down exponentially. At the 8 threads, the implementation time increases, but after that, it goes down.

“-n1 computation” doesn’t be affected by parallel programming little, because it’s computations is itself very small stuff. So, parallel programming is unuseful for this kind of small computation. But, in case of small things, the overhead affects on the computation. To increase scalability, that kind of overhead should be removed.

4.2

Actually I spent whole weekend. I was not familiar with pthead and OpenMP, so I studied by myself. It was a great experience.