ICSI 520 Distributed & Parallel Computing - Fall 2019

Homework 1

Santhosh Ranganathan (NetID: SR582413)

sranganathan2@albany.edu

**Part 1:**

Results:

|  |  |  |
| --- | --- | --- |
| **N (**generateInput N**)** | **Naive Implementation**  **(microseconds)** | **Optimized Implementation**  **(microseconds)** |
| 4 | 1.9 | 1.9 |
| 16 | 65.8 | 38.1 |
| 64 | 4001.1 | 2225.2 |
| 256 | 291888 | 134868.7 |
| 1024 | 9683410.6 | 3499281.3 |

make Part\_1 MAT\_SIZE=N

srun -n1 -c1 Part\_1\_naive.out

srun -n1 -c1 Part\_1\_optimized.out

(Averaging 10 runs)

Explanation:

To optimize the naive implementation of matrix optimization. I first considered how 2D arrays are stored in C which is row-major i.e. each row is stored contiguously in the memory. In the inner loop

C(i,j) = C(i, j) + A(i, k) + B(k, j)

I made a temporary array of size N (the size of square matrix) and used a for loop to store column values of B(k,j) before using them to calculate C(i,j) and a further small improvement was made using a temporary value to store the sum and assigning it to C(i,j) after going through all elements in A. Thus every element in A is accessed contiguously in the inner loop while requiring a limited number of temporary variables (1 + N). This reduced the frequency of cache misses thus improving overall speed.

**Part 2:**

Results:

|  |  |  |  |
| --- | --- | --- | --- |
| **Number (summing upto this number)** | **#Times executed (# of threads) p** | **Serial execution time (microseconds)** | **Parallel execution time (microseconds)** |
| 1000 | 2 | 58.5 | 187.9 |
| 10000 | 2 | 193.2 | 286.4 |
| 100000 | 2 | 1572.8 | 978 |
| 1000000 | 2 | 15229.4 | 7847.6 |
| 1000 | 8 | 104.4 | 616.6 |
| 10000 | 8 | 656.6 | 794.2 |
| 100000 | 8 | 6135.6 | 2013.8 |
| 1000000 | 8 | 60637.8 | 15717.4 |
| 1000 | 16 | 162.2 | 1027.8 |
| 10000 | 16 | 1268.6 | 1081.2 |
| 100000 | 16 | 12222.4 | 1423.4 |
| 1000000 | 16 | 121309.6 | 9527.8 |
| 1000 | 32 | 291 | 2159.4 |
| 10000 | 32 | 2484 | 2178.6 |
| 100000 | 32 | 24326.2 | 2683 |
| 1000000 | 32 | 242308.8 | 10836.8 |

make Part\_2

srun -n1 -c1 Part\_2\_serial.out 1000 2

srun -n1 -c2 Part\_2\_parallel.out 1000 2

(Averaging 5 runs)

Comparison:

For lower summations (number), the overhead is relatively greater and thus parallel execution is slower but as number increases exponentially, overhead matters less and parallel execution becomes faster. Also serial execution time increases proportional to increase in number.

**Part 3:**

Results:

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **P** | **Serial execution time (microseconds)** | **Parallel execution time (microseconds)** |
| 64 | 2 | 48.2 | 263 |
| 256 | 2 | 737 | 1135 |
| 512 | 2 | 2841.8 | 2370 |
| 1000 | 2 | 10756.6 | 6517 |
| 64 | 8 | 48.2 | 758 |
| 256 | 8 | 737 | 1498.4 |
| 512 | 8 | 2841.8 | 2254 |
| 1000 | 8 | 10756.6 | 4540.2 |
| 64 | 16 | 48.2 | 2267.8 |
| 256 | 16 | 737 | 2011 |
| 512 | 16 | 2841.8 | 2782 |
| 1000 | 16 | 10756.6 | 3228.2 |
| 64 | 64 | 48.2 | 6107.2 |
| 256 | 64 | 737 | 5943.4 |
| 512 | 64 | 2841.8 | 6136.2 |
| 1000 | 64 | 10756.6 | 6342 |

make Part\_3

srun -n1 -c1 Part\_3\_serial.out 1000

srun -n1 -c16 Part\_3\_serial.out 1000 16

srun -n1 -c36 Part\_3\_parallel.out 1000 64 (Max)

(Averaging 5 runs)

Explanation:

For low N, parallel is slower due to thread overhead but as N increases parallel becomes significantly faster. For 64 threads, only a maximum 36 core node was obtainable and in this case the overhead is significant and thus irrespective of N value, the execution time is roughly the same.