CSCI 654 – Advanced Computer Architecture
Homework 5

Due: December 2, 2016

Alexander Powell

3. (a) Translating from the openMP model to the message passing interface was relatively straightforward for this problem. It still remains a relatively simple reduction problem, so the same transformations that were applied in the previous homework can be used here. It makes it easier to transform the while loop into a for loop that simply iterations a large number of times (I used 10000000).

   (b) The table below shows the execution times of the algorithm above with the number of threads ranging from 1 to 16. The times listed are in microseconds.

# Threads	Exec Time	Speedup
1	1019660	1
2	559325	1.82
3	987892	1.03
4	312119	3.27
5	267083	3.82
6	233869	4.36
7	215494	4.73
8	189464	5.38
9	178318	5.72
10	163693	6.23
11	151630	6.72
12	141509	7.21
13	148673	6.86
14	126182	8.08
15	119416	8.54
16	115486	8.83

   From the results, it's clear that this program benefits greatly from thread level parallelism. By using 16 threads, we are able to achieve a speedup of almost 9 times the original time.

4. (a) 1000×10000 **inner loop**

Number of Threads	Time (microseconds)	Speedup
1	40448	1
2	13219	3.06
3	12288	3.29
4	13495	2.99
5	13895	2.91
6	13480	3
7	9162	4.41
8	6328	6.39
9	14176	2.85
10	15341	2.63
11	9415	4.3
12	10915	3.71
13	7906	5.12
14	7086	5.71
15	8839	4.58
16	5164	7.83

10000×1000 **inner loop**

Number of Threads	Time (microseconds)	Speedup
1	52732	1
2	18947	2.78
3	17891	2.95
4	14386	3.67
5	10888	4.84
6	13789	3.82
7	12002	4.39
8	11500	4.59
9	12580	4.19
10	9442	5.58
11	10866	4.85
12	8764	6.02
13	9268	5.69
14	13178	4
15	12344	4.27
16	13310	3.96

(b) 1000×10000 **outer loop**

Number of Threads	Time (microseconds)	Speedup
1	52295	1
2	12136	4.31
3	9124	5.73
4	10277	5.09
5	8094	6.46
6	11967	4.37
7	6952	7.52
8	8726	5.99
9	17999	2.91
10	26035	2.01
11	18741	2.79
12	20228	2.58
13	20834	2.51
14	27122	1.93
15	33514	1.56
16	31840	1.64

10000×1000 **outer loop**

Number of Threads	Time (microseconds)	Speedup
1	28943	1
2	24691	1.17
3	26593	1.09
4	19059	1.52
5	17043	1.7
6	16490	1.76
7	17059	1.7
8	11479	2.52
9	13369	2.16
10	9232	3.14
11	7344	3.94
12	4345	6.66
13	3799	7.62
14	5687	5.09
15	2264	12.8
16	4557	6.35

(c) From observing the results it's clear that adding parallelism greatly improves the overall performance. In general it seems like a slightly better speedup was attained when computing the 10000×1000 matrix as opposed to the 1000×10000 matrix. This can probably be attributed to the impact of data locality. Also, it should be noted that some rather odd speedups were found in a few of the calculations, where a higher number of threads actually resulted in a lesser speedup. This is probably more of a result of added overhead to the code than the threading itself. It should be noted that the formula used for speedup is $\frac{\text{Time}_{\text{orig}}}{\text{Time}_{\text{threaded}}}$.

5. (a) Similarly to homework 4, this code was broken up into three separate loops in a manner so as to avoid dependency conflicts. The first loop keeps a running total of A[i] in a variable and on every iteration sets the value of A[i] to be the current value of that running total at that time. From there the cumulative sum is calculated in parallel.

   (b) When using openMP, the overhead of adding a threaded structure to the code was not amortized until the size of A reached 1 million. In the case of MPI, the results show that adding multiple threads benefits the performance when the size of A is 100000. In each case 8 threads were used and times are given in microseconds.

N	**Sequential Time**	**Threaded Time**
10	90	297
100	75	301
1000	73	296
10000	212	318
100000	1259	511
1000000	13300	2313
10000000	81880	19921
100000000	639327	143175

Note on CS department nodes:

In my experiments I used the department nodes *pepe*, *al* and *squidward*. When running the command

```
grep -c ^processor /proc/cpuinfo
```

it tells me that there are 8 CPUs/cores on each of these machines, so some experiments only used one of these nodes, while others used all of them.