Spring 2018 CSE613 HW1

Abiyaz Chowdhury
StonyBrook ID: 111580554

March 24, 2018

Part 1 (Comet was used for all calculations)

A

Value of r	Time for 1st(ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	6th (ms)
1	0	$4.1 \cdot 10^{-4}$	0	0	0	$4.1 \cdot 10^{-4}$
2	$4.1 \cdot 10^{-4}$	$4.1 \cdot 10^{-4}$	$4.1 \cdot 10^{-4}$	$8.21 \cdot 10^{-4}$	$8.2 \cdot 10^{-4}$	$4.11 \cdot 10^{-4}$
3	$2.87 \cdot 10^{-3}$	$2.46 \cdot 10^{-3}$	$2.46 \cdot 10^{-3}$	$2.46 \cdot 10^{-3}$	$2.46 \cdot 10^{-3}$	$2.87 \cdot 10^{-3}$
4	$1.85 \cdot 10^{-2}$	$1.81 \cdot 10^{-2}$	$1.81 \cdot 10^{-2}$	$1.89 \cdot 10^{-2}$	$1.81 \cdot 10^{-2}$	$1.89 \cdot 10^{-2}$
5	0.14	0.14	0.14	0.15	0.14	0.15
6	1.16	1.15	1.16	1.17	1.12	1.18
7	11.16	10.39	36.51	26.78	9.42	43.77
8	171.97	133.98	161.32	200.06	156.6	195.86
9	1,526.22	1,077.05	1,381.47	1,404.45	1,134.08	1,445.69
10	15,781.1	9,418.85	15,274.8	21,739.4	8,638.92	19,988.8
11	$1.72 \cdot 10^{5}$	72,153.9	$1.5 \cdot 10^{5}$	$2.1 \cdot 10^{5}$	77,590.3	$1.8 \cdot 10^{5}$

B

Implementation	L1 misses	L2 misses	L3 misses
1	$4.3 \cdot 10^{9}$	19	31
2	$4.3 \cdot 10^{9}$	5	30
3	$4.3 \cdot 10^{9}$	9	38
4	$4.3 \cdot 10^{9}$	11	32
5	$4.3 \cdot 10^{9}$	5	30
6	$4.3 \cdot 10^{9}$	12	33

C

The 2nd and 5th implementations run the fastest. This is because they have less cache misses, since arrays are stored in C++ in row-major order, locality of reference is obtained when accessing entries that are on the same row. The inner-most loop should be j, since this results in the fewest cache misses. Also, observe that all implementations have the same number of L1 accesses/misses, but implementations 1 and 4 (whose innermost loop is j) result in the most number of L2 cache misses.

D

The correct implementations (that do not introduce race conditions) for both implementations *ikj* and *kij* are i, ij, j. (Note: Values shown as 0 took too long to run and were terminated before completing).

s	ikj (i)	ikj (ij)	ikj (j)	kij (i)	kij (ij)	kij (j)
4	0.15	0.36	1.21	$8.48 \cdot 10^{-2}$	0.3	1.21
5	$7.18 \cdot 10^{-2}$	0.97	4.84	0.2	1.09	4.67
6	0.47	4.03	18.84	0.77	4.31	18.79
7	3.33	18.27	76.15	3.93	19.5	73.93
8	26	90.1	303.29	28.99	93.95	311.39
9	0	199.81	0	0	212.73	0
10	0	1,590.99	0	0	1,681.79	0
11	0	12,694.7	0	0	13,404.2	0

E

Cores	Time(ms) ikj	Time(ms) kij
1	50,701.7	53,864.7
2	25,287.9	26,889.3
3	16,880.9	17,946.6
4	12,647.1	13,445.9
5	10,125.4	10,765.5
6	8,445.87	8,981.02
7	7,235.26	7,695.93
8	6,322.61	6,725.79
9	5,628.83	5,990.59
10	5,059.5	5,386.4
11	4,615.91	4,913.14
12	4,221.21	4,496.16
13	3,900.21	4,154.71
14	3,628.86	3,867.01
15	3,380.97	3,605.4
16	3,159.69	3,370.28
17	2,987.05	3,186.91
18	2,814.09	3,004.23
19	2,666.09	2,847.3
20	2,543.17	2,715.95
21	2,434.31	2,616.32
22	2,332.33	2,505.83
23	2,223.74	2,395.51

F

Some of the parallelizations didn't work because they introduce race conditions which corrupts the final result. Parallelizing too many for loops leads to too much overhead, which can negate the benefits of parallelization if enough processor cores are not available or if the ratio of work needed for intercommunication between the processors is great in comparison to the actual work done by the processors.

G

These results were computed for $r = 11$, as obtained in part A. For very low values of m, the algorithm took prohibitively long to run, which indicates the high cost of too much overhead. Interestingly, the values don't change much for higher values of m, although lower values took considerably longer to run.

Value of m	Time(ms)
8	$1.87 \cdot 10^5$
16	$1.13 \cdot 10^5$
32	93,139.3
64	93,296.2
128	98,316
256	96,057.1
512	92,498.2

Part 2 (Comet was used for all calculations)

A

s	Time (ms) (work-steal)	GFLOPS
4	4.6	$1.78 \cdot 10^{-3}$
5	4.65	$1.41 \cdot 10^{-2}$
6	6.96	$7.53 \cdot 10^{-2}$
7	10.76	0.39
8	58.09	0.58
9	103.38	2.6
10	720.7	2.98
11	3,857.39	4.45
12	28,590.7	4.81
13	$2.29 \cdot 10^5$	4.81

s	Time(ms) work-share	GFLOPS
4	1.69	$4.85 \cdot 10^{-3}$
5	0.39	0.17
6	2.65	0.2
7	23.78	0.18
8	348.8	$9.62 \cdot 10^{-2}$
9	452.81	0.59
10	7,712.59	0.28
11	40,377.3	0.43
12	$3.08 \cdot 10^{5}$	0.45

We note that work-stealing performs better than work-sharing for larger matrices, probably due to the fact that work-sharing involves a lot of time spent doing nothing when a thread's deque is empty. It basically has to just wait and hope that another thread gives it some work.

B

C

Num cores	Time (ms) work-steal	Efficiency
1	85,509.1	1
2	42,698.1	1
3	28,511.2	1
4	21,771.8	0.98
5	18,577.5	0.92
6	29,649.1	0.48
7	12,571.8	0.97
8	13,096.4	0.82
9	9,500.91	1
10	8,580.44	1
11	7,816.95	0.99
12	7,147.51	1
13	6,615.42	0.99
14	6,140.92	0.99
15	5,794.11	0.98
16	5,366.68	1
17	5,077.56	0.99
18	4,802.61	0.99
19	4,516.84	1
20	4,318.93	0.99
21	4,251.85	0.96
22	3,958.85	0.98
23	3,859.63	0.96

Part 3

A

This is similar to the coupon collector's problem from probability theory. After $n-1$ different deques have been visited, the probability that a processor will visit one of the remaining $p-(n-1)$ deques is $(p-(n-1))/p$. We can define t_n as the number of steal attempts needed to steal from a new deque after we have stolen from $n-1$ different deques. This is a geometric distribution, and if we have already visited $n-1$ deques, the expected number of attempts before a new deque is visited is the inverse of the probability, i.e. $p/(p-(n-1))$. The expected number of attempts needed to visit all the deques is therefore $E[t_1]+E[t_2]+...+E[t_n] = p/p+p/(p-1)+...+p/1 = p(1+1/2+...+1/p) = p*O(log\ p)$.

B

It is possible for a thread to visit all the deques after some number of consecutive attempts and find that they are empty, but it is possible that one of the deques visited earlier is given more work after it has already been visited. Since the thread doesn't visit all the deques within a single attempt, its information from previous attempts is not always up to date.

C

Every task, including the root task, must eventually be executed, since the leaf nodes must be on some thread, all of which must eventually execute, thereby completing the root node's work. It is possible for some threads to terminate early, but other threads will still have to do the remaining work.

D

This is similar to the balls into bins problem from probability theory.

E