COMP60611 Laboratory Exercise 3: Simple BLAS-like examples Solutions

Team:

Gold- Group 5

Team Members:

Christos Theofilou

Zijiang Yang

Yini Zhang

Kai Shen

Part 1

Question a

Table 1 Sequential Timing Results

Operation Routine	1	2	3
Vector update time (seconds)	1.8089	1.8088	1.8066
Vector add time(seconds)	0.3457	0.3474	0.3474
Sum time	0.1123	0.1123	0.1124
Matrix initialisation time(seconds)	2.4370	2.1012	2.1094
Trimv time(seconds)	1.1191	1.0790	1.0797
Total time(seconds)	5.8239	5.4496	5.4564

We observed the matrix initialisation time in first run was a bit larger than that in the next two runs.

Question b

Table 2 Parallel Timing Results

Cores	1	2	3	4	6	8	12	24	36
Vector update time(seconds)	1.8089	1.8357	1.8726	1.8565	1.8677	1.8575	1.8677	1.8777	1.9109
Vector add time(seconds)	0.4272	0.6071	0.4288	0.3483	0.2873	0.2779	0.3039	0.3375	0.4315
Sum time(seconds)	0.1151	0.0605	0.0421	0.0330	0.0251	0.0216	0.0211	0.0238	0.0549
Matrix initialisation time(seconds)	0.2435	0.4405	0.4547	0.4142	0.4166	0.3861	0.3917	0.4039	0.4138
Trimv time (seconds)	0.8721	1.4422	1.5821	1.7693	2.0278	2.2362	2.2970	2.2087	2.1183
Total time (seconds)	3.4678	4.3871	4.3815	4.4227	4.6258	4.7806	4.8831	4.8531	4.9310

The numbers used above were based on an average of 3 runs.

Question c

Table 3 Timing Results of Vector Addition Operation

Cores	1	2	3	4	6	8	12	24	36
Vector add time(seconds)	0.3859	0.1958	0.1326	0.1012	0.0706	0.0569	0.0405	0.0344	0.0347

For this question we have also plot the graph of performance and it can be seen in Appendix A.

Question d

Initially, array x[], a[], b[] only exist in the memory close to one of the processors since mcore48 is a NUMA DSM multiprocessor. When threads send the first read or write (load or store) requests, the cores cannot find a copy in local caches(L1, L2, L3) which means a miss happens. Then the core will make a copy of the 64KB data in the memory which contains the data that the threads are interested in to fill one cache line in all the three levels of caches(if the request is a store the core will write to the cache line instead of the memory). When the subsequent requests come, the core will check the caches if they have the copies. If it a miss happens, the core will repeat the similar behaviour above. If a hit happens, the core will read from or write to the cache instead of the memory. As the progress running, all the cores will have a copy of the latest data in a[] in their L2 caches considering that the L1 caches do not have enough space while L2 caches may have. At last, if the master thread wants to

print the results while the memory does not have the latest data, the master thread will find and move the data to its cache and then print out the results.

Question e

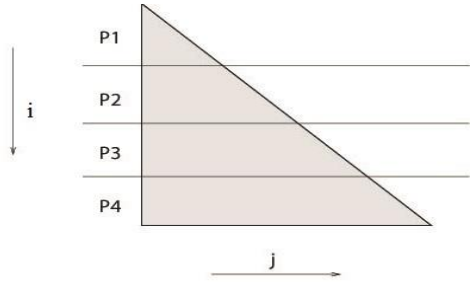

Figure 1 load imbalance

In Figure 1 we can see how the block partitioning is separated. We can observe that the load of tasks is not equally separated and for example P4 has more load of work than P1. In the code each iteration of the i loop requires 2i -1 floating point operations which leads to load imbalance.

Question f

Table 4 Cyclic partitioning strategy

Threads	1	2	3	4	6	8	12	24	36
Matrix initialisation time(seconds)	0.4957	1.0924	1.1070	1.2369	1.1687	1.2237	1.2135	1.1899	1.1440

The Matrix initialisation time from the Cyclic partitioning strategy should have been less than when we parallelised the vector update routine, but we can observe that this is not happening. (e.g. For 6 threads in question C the time is 0.4105 seconds and for 36 threads is 0.4104 seconds).

Question g

Table 5 Subroutine trimv

Threads	1	2	3	4	6	8	12	24	36
Trimv time (seconds)	0.8726	3.1791	5.8911	6.8134	8.1680	8.7077	9.9210	13.1153	16.3931

The code is effectively equivalent since the y[m] and yy[m] [1] fill the cache line in the same way.

In question e the trimv time for 6 threads is 2.0632 seconds compared to 8.1680 seconds. In question f the trimv time for 6 cores is 2.3830 seconds. We have also plotted the graph of performance and it can be seen in Appendix B.
Considering that our cache line is 64 bytes and the size of each element in the array is 8 bytes, there are 8 elements in each cache line. When the array is yy[m][8], different threads will use different cache lines as each cache line has one element that can be used and the other seven elements will be null(Question e and f). When the array changes into yy[m][1], each cache line will be filled with 8 elements as well but all the elements can be used from the threads, so every different thread will be trying to use the same cache line at the same time. As a result, there will be time consumed due to other threads trying to use the same cache line.

Question h

The capacity of array a[] is 1024*60*8 bytes, as well as b[] and x[]. Array y[] and l[] will each occupies 512*8 bytes. So the memory requirement of the code is (3*1024*60+2*512)*8 =1482752 bytes = 1448Kb(when 1 Kb = 1026 bytes). We can know the capacity of L1 cache and L2 cache are 64Kb and 512Kb respectively and the capacity of L3 cache is 5118Kb, which means the memory requirements of this code do not exceed the capacity of the level 3 cache in the case of single processor execution.

Question i

As verified before, level 3 cache is enough for the processor execution. It would not take too much time to transfer data between cores and memories. We thought that the scattered version may cost a little more time for that it would take more time to transfer data between different chips while the compact version runs on the same chip. But the result did not match our speculation. The time costs of the two version on vector updating were the same. So were the vector add time and sum time. But time cost of compact version on matrix initialisation time was double that of the scattered version., so was the trimv time. We speculated that in the compact version, requests and responses were all from and sent to the same chip. As a result, the bus became more congested. What was more, matrix operation shared the cache line less often than the vector operation which increased the frequency of data transferring and made the bus even more congested.

Table 6 Timing results using four threads on four cores (m = 512)

	compact		scattered	
	Time(s)	Standard deviation	Time(s)	Standard deviation
vector update time	0.4123	0.0063	0.4071	0.0007
vector add time	0.1036	0.0012	0.1042	0.001
sum time	0.03187	9.17E-05	0.0326	2.8361E-05
matrix initialisation time	2.0801	0.1397	1.0949	0.0086
trimv time	3.8698	0.1229	2.1489	0.0316
Total time	6.4984	0.02135	3.7883	0.0401

Question j

Because m is tripled, the size of data needing processed is approximately nine times the previous version of which the result has been approved. The cost of data transferring in the scattered version is more significant since the data has to travel further.

Table 7 Timing results using four threads on four cores (m = 1536)

	compact		scattered	
	time(s)	Standard deviation	time(s)	Standard deviation
vector update time	0.4223	0.0031	0.4129	0.0065
vector add time	0.1027	0.0005	0.1027	0.0008
sum time	0.03216	6.001E-05	0.0327	4.7466E-05
matrix initialisation time	16.7553	0.6662	9.5825	0.0235
trimv time	34.0882	1.864	18.4356	0.0697
Total time	51.402	1.196	28.5672	0.0742

Question k

Time cost of vector updating in compact version and scattered version both decreased a little in the backwards loop situation. We speculated this was because of cache line alignment. On the other hand, vector add time cost increased a lot because after the backwards loop in vector updating, the data had been moved into different cores' caches. In this case, the threads will move the data from other cores and cost more time. In question I, however, they had no need to do this because vector updating and adding functions in the same thread process the same part of the data. The scattered version even costs more time for transferring data between chips (the compact version transfer data within one chip).

Table 8 Timing results using four threads on four cores (backwards)

	compact		scattered	
	time(s)	Standard deviation	time(s)	Standard deviation
vector update time	0.3806	0.015	0.3818	0.0034
vector add time	0.2089	0.0066	0.2931	0.0007
sum time	0.032	5E-05	0.0327	0.0001
matrix initialisation time	2.2394	0.062	1.1055	0.0135

trimv time	3.8394	0.054	2.1292	0.0368
Total time	6.7011	0.0569	3.943	0.0413

The numbers used in all the questions were based on an average of 3 runs.

Devision of work

Christos and Zijiang finished 7 of 11 questions and Kai and Yini finished the left.

Appendix

Appendix A

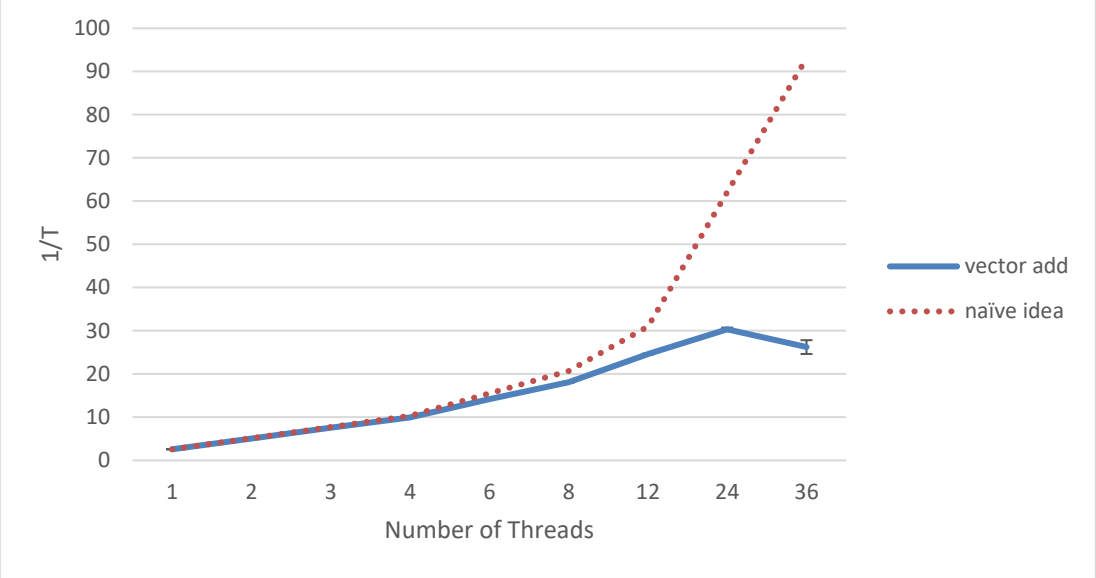

Figure 2 performance of the vector addition

Appendix B

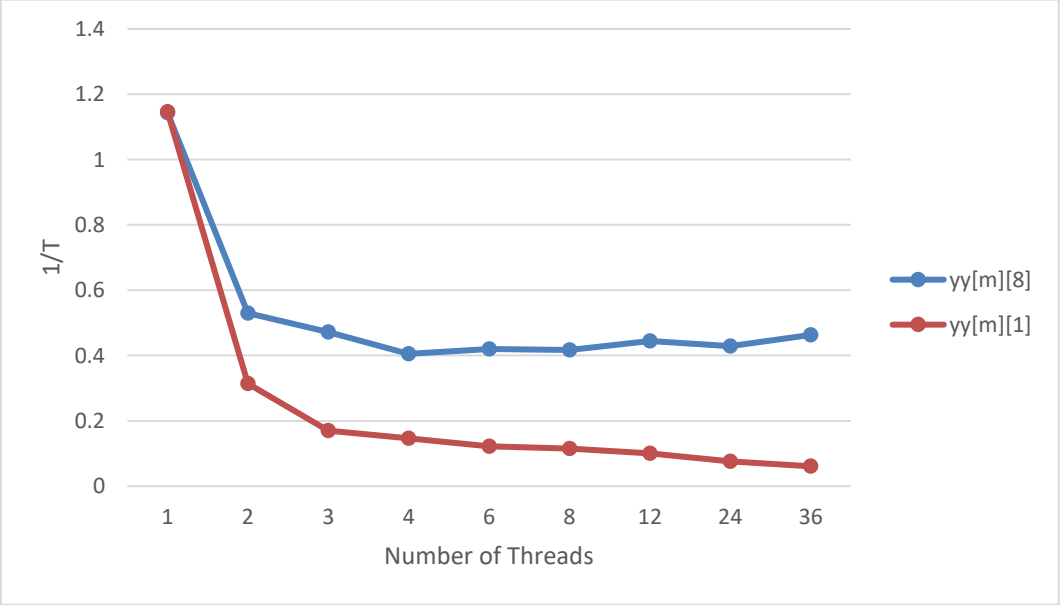

Figure 3 performance of the subroutine trimv

To make the difference between yy[m][1] and yy[m][8] more clear, we did not plot naïve idea in this figure.