CA670 Concurrent Programming

Name	Aruna Bellgutte Ramesh
Student Number	18210858
Programme	MCM (Cloud Computing)
Module Code	CA670
Assignment Title	Parallel Sum Reduction in OpenMP and OpenCL
Submission date	22nd April 2019
Module coordinator	David Sinclair

Name: Aruna Bellgutte Ramesh Date: 22nd April 2019

Parallel Sum Reduction in OpenMP and OpenCL

Problem Statement

To implement efficient solution for Parallel Sum Reduction that computes sum of very large arrays in both OpenMP and OpenCL framework.

Solution

OpenMP

Solution 1: Normal Algorithm

In this solution, the program is implemented such that a simple linear summation of the array values is done while traversing through the loop. Like:

```
for(long i=0;i<veryLongArraySize;i++)
{
        sum = sum + veryLongArray[i];
}
```

This when computed serially/sequentially, takes a long time. For example, for an array size of 2^29, a serial summation takes about 1.477690 seconds. However, when computed parallelly using OpenMP directives, this time reduces.

Parallel Directives Used:

1. OpenMP parallel for construct

   OpenMP parallel for construct was used to parallelize the summation. It was observed that, for a typical large array with array size greater than or equal to 2^20, the time taken to execute the program using a parallel for construct is less than serial execution of the same. For example, for an array size of 2^29, a parallel summation with parallel for construct takes about 0.672336 seconds only. However, when compared between parallel for construct and parallel task construct, we can find some interesting findings which will be discussed under the parallel task construct. Table 1 shows the results for the same.

2. OpenMP parallel task construct

   Along with OpenMP parallel for construct, OpenMP parallel task construct too was used to parallelize the summation. Results of which are tabulated in Table 1. Like parallel for construct, parallel task construct too reduces the time of summation when array size is greater than or equal to 2^20. For example, for an array size of 2^29, a parallel summation with parallel task construct takes about 0.599212 seconds only. Typically parallel task constructs are more efficient than parallel for construct. But for array sizes greater than or equal to 2^30, parallel task construct degrades in performance compared to parallel for construct.

Table 1. OpenMP Normal Algorithm Results

Test Case IDs	Size of array	Size of array (expressed in powers of 2)	Time taken (in seconds)		
			Parallel Execution		Serial Execution
			Loop Construct	Task Construct	
1.	1024	2^10	0.001041	0.000691	0.000007
2.	131072	2^17	0.001840	0.000928	0.000716
3.	262144	2^18	0.001406	0.000641	0.001381
4.	524288	2^19	0.001704	0.002963	0.001701
5.	1048576	2^20	0.002723	0.001414	0.005326
6.	268435456	2^28	0.366683	0.315524	0.762889
7.	536870912	2^29	0.672336	0.599212	1.477690
8.	1073741824	2^30	9.028112	11.829503	15.061498

Screenshots of these test cases are in screenshots folder.

Solution 2: Parallel Sum Reduction Algorithm

In this algorithm, the program is implemented such that rather than a simple linear summation of array values, the summation is done as in the below algorithm:

Step 1: A "slider" value is calculated as (veryLongArraySize/2).
Step 2: An array value at (i)th location is added with an array value of (i+slider)th location and stored back to (i)th location. Now half the array values are processed stored. So now this half should be processed again.
Step 3: Reduce array size, veryLongArraySize, by 2 and continue with step 1. If array size is 1 go to step 4.
Step 4: Value at veryLongArray[0] gives the sum of the very large array values.

Diagram 1 below explains this algorithm. This algorithm was implemented in OpenMP framework. This program shows the results for normal linear summation, parallel sum reduction algorithm without any OpenMP constructs and parallel sum reduction algorithm with parallel for construct. The results of which are tabulated in table 2.

It can be observed from the results that **parallel sum reduction algorithm is not suited for OpenMP framework.** The results shows that the algorithm takes more time than serial summation. However, this program doesn't scale well for vary large arrays above 2^22 array size. This is because of the static allocation of 2 dimensional array. Result[SIZE][SIZE] where SIZE is veryLongArraySize couldn't be done without using malloc and I have not used malloc. But typically, it only need Result[log(SIZE)/log2][SIZE]. But logarithm can't be calculated at compile time hence couldn't implement for very large array sizes.

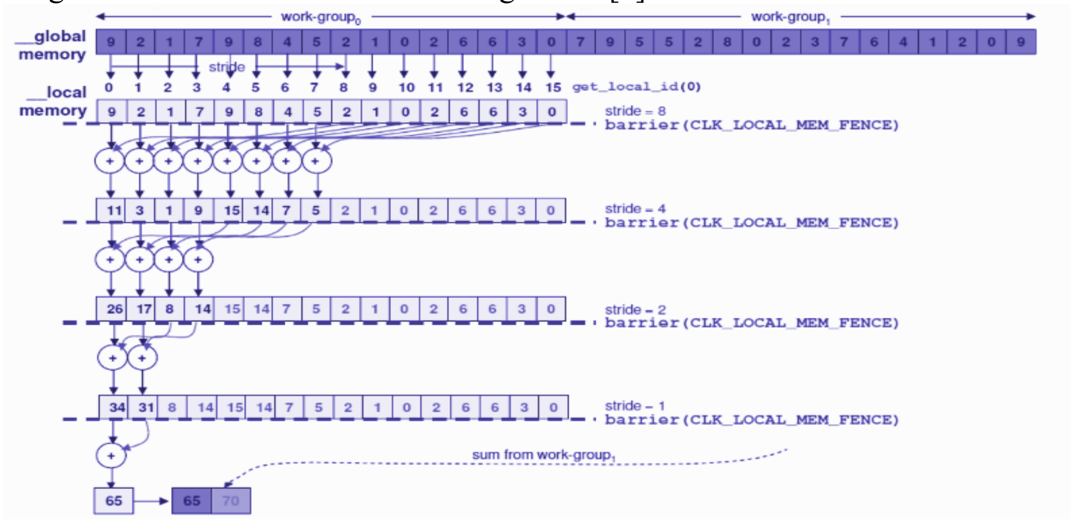
Table 2: OpenMP Parallel Sum Reduction Algorithm Results

Test Case IDs	Size of array	Size of array (expressed in powers of 2)	Time taken (in seconds)		
			Serial Execution		Parallel Execution using Parallel Sum Reduction
			Normal Algorithm	Parallel Sum Reduction Algorithm	

					Algorithm and Loop construct
1.	1024	2^10	0.000004	0.000052	0.001398
2.	1048576	2^20	0.003000	0.015744	0.004023
3.	2097152	2^21	0.006220	0.033298	0.009081
4.	4194304	2^22	Program doesn't work due to allocation of memory for the number array at compile time.		
5.	1073741824	2^30			

Screenshots of these test cases are in screenshots folder.

Diagram 1: Parallel Sum Reduction Algorithm [1]



OpenCL

<u>Solution:</u> Parallel Sum Reduction Algorithm in OpenCL

The Parallel Sum Reduction Algorithm, explained above, is best suited for OpenCL framework. The algorithm was implemented with WorkerItems equal to the size of very large array. GroupSize was set to 256. Also, GroupSize was evenly dividing WorkerItems. The results of this algorithm as run on GPU is tabulated in table 3.

It can be observed from the results that for small array sizes, less than 2^20, parallel execution takes longer than serial execution. But for sizes above and equal to 2^20, parallel execution takes shorter time than serial execution. The results exponentially get better as the array size increases. For example, for array size 2^30, serial execution took 15.65 seconds but parallel execution took only 0.53 seconds. These results are way better than OpenMP framework. However, OpenCL framework comes with an overhead of setting up the environment and setting up the kernel. **This sets off the advantage over OpenMP**. May be for array sizes **way too greater than 2^30** and the **right hardware**, OpenCL will be better suited than OpenMP for summation of very large array values. However, this cannot be verified in our local systems and hence is an open ended observation.

Table 3: OpenCL Parallel Sum Reduction Algorithm Results

Test Case IDs	Size of array	Size of array (expressed in powers of 2)	Time taken (in seconds)	
			Parallel Execution	Serial Execution

1.	1024	2^10	0.519	0.3
2.	131072	2^17	0.946	0.303
3.	262144	2^18	0.1080	0.628
4.	524288	2^19	0.1453	0.1775
5.	1048576	2^20	0.2129	0.3784
6.	8388608	2^23	0.8261	0.21359
7.	134217728	2^27	0.69504	0.346805
8.	268435456	2^28	0.147371	0.645622
9.	536870912	2^29	0.281402	4.36178
10.	1073741824	2^30	0.530579	15.648802

Screenshots of these test cases are in screenshots folder.

Conclusion / Justifications

Owing to the test cases (as tabulated in tables) and screenshots attached, the study can be summarized as below:

1. For OpenMP framework and very large array sizes, above 2^30 array size, Parallel For construct is best.
2. For smaller array sizes, between 2^20 and 2^30 array size, OpenMP Parallel Task construct is better.
3. Parallel Sum Reduction Algorithm is best suited for OpenCL framework. Implementing the algorithm on OpenMP framework only degrades the results.
4. OpenCL framework gives better results than OpenMP, for computing summation of very large arrays, typically above 2^30 size. But it comes with the cost of environment setup time.
5. May be for array sizes **way too greater than 2^30** and the **right hardware**, OpenCL will be better suited than OpenMP for summation of very large array values. However, this cannot be verified in our local systems and hence is an open ended observation.

References

[1] "Parallel Reduction of Sum - GPU / OpenCL versus CPU," [Online]. Available: https://dournac.org/info/gpu_sum_reduction.

[2] T. Mattson. [Online]. Available: https://www.youtube.com/watch?v=nE-xN4Bf8XI&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG&index=1.