**Name:** Ruben Vasconcelos

**Student Number:** 16212630

**Module Code:** CA670

**Assignment Title:** Parallel Sum Reduction in OpenMP and OpenCL

**Submission Date:** 10am on Monday 24th April 2017.

**Plagiarism Declaration:**

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

Ruben Vasconcelos

23-April-2017

I was unable to run openMP code on my personal machine, so I used one of the university’s lab computers to develop and test the OpenMP solution.

While for my OpenCL implementation, I used my personal machine, because it’s a Mac-book and the lecturer provided an example on how to set it up in Xcode.

**OpenMP:**

**Environment information:**

CPU: Intel® Core™ i5-6500 CPU @ 3.20GHz × 4

Memory: 7.7 GiB

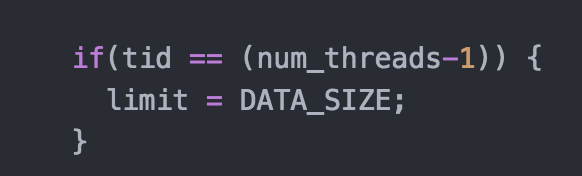
OS: openSUSE Tumbleweed 64-bit

**Algorithm Description:** **2 stage reduction**

Create n threads and partition the input array into the same number of blocks, then perform a sequential sum reduction on each block.

Each thread is given a local copy of sum (initially 0), after each thread finishes their task, a second reduction is performed in order to add their local copy of to the global value of sum using “reduction (+: sum)”.

In order to account for the times when the size of the array is not divisible by the number of threads an extra check is put in place.



This means one of the threads might have to do a bit of extra work but it would be minimal, and not enough to impact the general performance.

**Comparison methods:**

In order to prove that my implementation is efficient, I decided to compare it against two different solutions. The first comparison method is simply a sequential sum, the second method uses the OpenMP sum reduction clause to perform the reduction on the whole input array.

|  |  |
| --- | --- |
| ComparisonMethod-1 | ComparisonMethod-2 |
|  |  |

**Results:**

**Input size:** 1,003 items

**Number of Threads:** 4

|  |  |  |
| --- | --- | --- |
| **2 Stage Reduction** | **Comparison Method-1** | **Comparison Method-2** |
| 312 microseconds  1499 microseconds  154 microseconds  331 microseconds  162 microseconds  154 microseconds  154 microseconds  148 microseconds  2736 microseconds  693 microseconds | 9 microseconds  9 microseconds  9 microseconds  9 microseconds  9 microseconds  10 microseconds  9 microseconds  9 microseconds  9 microseconds  8 microseconds | 165 microseconds  485 microseconds  178 microseconds  1290 microseconds  155 microseconds  160 microseconds  147 microseconds  163 microseconds  333 microseconds  153 microseconds |

I intended to show simply an average of the time taken by each approach, but as you can see there were quite a few spurious results hence the reason why I decided to show the results of 10 test runs in the table above.

As the input array became larger, I stopped noticing the behavior seen above which allowed me to use the averages.

For small input sizes the sequential method performed a lot better than the other two, this was expected as creating threads comes with a certain cost.

**Input size:** 100,000 items

**Number of Threads:** 4

|  |  |  |
| --- | --- | --- |
| **2 Stage Reduction** | **Comparison Method-1** | **Comparison Method-2** |
| Avg. 133.6 microseconds | Avg. 255.5 microseconds | Avg. 146.5 microseconds |

**Input size:** 500,001 items

**Number of Threads:** 4

|  |  |  |
| --- | --- | --- |
| **2 Stage Reduction** | **Comparison Method-1** | **Comparison Method-2** |
| Avg. 333.4 microseconds | Avg. 1023.9 microseconds | Avg. 402.9 microseconds |

At this point, it was quite clear the comparison method-1 performance was nowhere near the other two approaches. And as the size of the input array grew the difference just grew larger.

**Input size:** 1,000,000 items

**Number of Threads:** 4

|  |  |
| --- | --- |
| **2 Stage Reduction** | **Comparison Method-2** |
| Avg. 599.1 microseconds | Avg. 727 microseconds |

**Input size:** 2,090,152 items

|  |  |  |
| --- | --- | --- |
| Threads | **2 Stage Reduction** | **Comparison Method-2** |
| 1 | Avg. 3493.4 microseconds | Avg. 4327.8 microseconds |
| 2 | Avg. 1892.2 microseconds | Avg. 2340 microseconds |
| 4 | Avg. 1112.3 microseconds | Avg. 1345.8 microseconds |
| 12 | Avg. 1611.1 microseconds | Avg. 2193.4 microseconds |

Even though the two approaches performed very similar for smaller input arrays, the 2-stage reduction method implemented, performed increasingly better as the size of the input array grew larger.

Increasing the number of threads was only beneficial up to a certain point, in this case 4 as the machine used had 4 cores available.

**OpenCL:**

**Environment information:**

CPU: 2.9 GHz Intel Core i7

Memory: 16 GiB

OS: macOS Sierra 10.12.4

GPU: Radeon Pro 455 2048 MB

**Algorithm:** 2 stage reduction, same algorithm as the one explained in the OpenMP section, the only difference is that now the second reduction is performed with a for loop that iterates through the kernel output array from 0 up to the number of groups and sums each result.

**Comparison methods:** Onceagain I’ll be comparing the OpenCL implementation against the sequential method. For OpenCL picking the number of partitions necessary is not as simple as choosing the number of cores so I’ll also be comparing the two-stage method against itself by varying the number of partitions and items in each partition block

OpenCL average preparation time: 36,152 microseconds, performing all the setting up work necessary to execute the kernel commands.

**Input size:** 1 ,000 items

|  |  |  |  |
| --- | --- | --- | --- |
| Blocks | Items per block | **2 Stage Reduction** | **Sequential** |
| 1,000 | 1 | Avg. 992.1 microseconds | Avg. 2 microseconds |
| 100 | 10 | Avg. 998.2 microseconds |
| 1 | 1000 | Avg. 2249.3 microseconds |

**Input size:** 100,000 items

|  |  |  |  |
| --- | --- | --- | --- |
| Blocks | Items per block | **2 Stage Reduction** | **Sequential** |
| 100,000 | 1 | Avg. 3968.9 microseconds | Avg. 201.5 microseconds |
| 1,000 | 100 | Avg. 1532.1 microseconds |
| 100 | 1000 | Avg. 2191.4 microseconds |

At this point it’s becoming obvious that the number of block reductions and number of items in each block, drastically influence the outcome.

As expected the kernel executions are quite expensive which also has a massive impact in the time taken by the OpenCL program in comparison with the sequential algorithm.

**Input size:** 1,000,000 items

|  |  |  |  |
| --- | --- | --- | --- |
| Blocks | Items per block | **2 Stage Reduction** | **Sequential** |
| 5,000 | 200 | Avg. 2517.5 microseconds | Avg. 2030 microseconds |

**Input size:** 2,000,000 items

|  |  |  |  |
| --- | --- | --- | --- |
| Blocks | Items per block | **2 Stage Reduction** | **Sequential** |
| 80,000 | 25 | Avg. 4620.2 microseconds | Avg. 4173.2 microseconds |
| 10,000 | 200 | Avg. 4093 microseconds |
| 20 | 100,000 | Avg. 42154.7 microseconds |

As you can see highlighted above the OpenCL program performs increasingly better as the input array increases in size, you also need to take into account the average 36,152 microseconds taken to perform all the set-up work initially, so it would require quite large input array to make it worth it in this case.