Parallel Computing
K-means Algorithm

Laura Rodrigues
Masters in Software Engineering
University of Minho
Braga, Portugal
pg50542@alunos.uminho.pt

Mariana Rodrigues
Masters in Software Engineering
University of Minho
Braga, Portugal
pg50628@alunos.uminho.pt

***Abstract*—The K-means Algorithm takes a given number of points and groups them by clusters. As a result we get a set of cluster centroids that allow us to organize the points. Each point is closer to it's centroid than to any other centroid. In this paper, we implemented and analysed a parallelized version of this algorithm, using different resources.**

***Index Terms*—k-means, parallel, OpenMP, CUDA**

I. Introduction

The goal of this project was to optimize the previously developed parallel program using OpenMP directives or to explore new platforms in order to improve the parallelization of our program.

II. Implementations

As referred previously, in this assignment we intended to improve a parallel implementation of the algorithm kmeans. We started by looking into our previous implementation that used OpenMP.

III. Código Inicial

Firstly, we will start by analysing the program previously developed. The metrics we intend to analyse are the percentage of serial work, memory bandwidth and computational intensity (Memory Wall), task granularity and parallelism overhead, synchronisation overhead and compute time per parallel task. We believe that looking at this metrics, may reveal what can be improved in our solution.

#T	TExec	SpeedUp	#Ix10^9	#CCx10^9	#Mx10^7	CPI
1	3	1	20,17	8,35	8,53	0,4
4	1,53	2,19	20,23	15,7	8,56	0,8
8	0,86	3,88	20,4	14,37	8,56	0,7
12	0,66	5,09	20,26	15,93	8,57	0,8
16	0,64	5,20	20,26	15,9	8,58	0,8
20	0,59	5,71	20,28	16,4	8,58	0,8

#T	TExec	SpeedUp	#Ix10^9	#CCx10^9	#Mx10^7	CPI
1	17	1	113	49,82	9	0,43
4	6,69	2,55	112,87	63,9	8,9	0,6
8	4,26	4,01	113,08	82,19	9,06	0,7
12	3,04	5,61	113,24	80,87	9,18	0,7
16	2,50	6,82	114,06	98,53	9,22	0,87
20	2,26	7,57	115,26	101,57	9,27	0,9

Tables 1 & 2: Results obtained using different numbers of threads with 4 and 32 clusters respectively

Serial Work:
As stated in the previous report, some of our code could not be parallelized. That includes the creation of all the points and assignment of the first centroids. Besides that, the kmeans "while" cycle is sequential. The fraction of sequential code we maintain is:

$$Maximum\_speedup = \frac{1}{serial\_fraction\_of\_work}$$

From Amdahl's law: $speedup \leq \frac{1}{f+\frac{1-f}{p}}$

For K = 4 and T = 4: $3,89 \leq \frac{1}{f+\frac{1-f}{4}} \equiv$
$\equiv$ 3,89 $\leq \frac{1*4}{4f+(1-f)}$
$\equiv$ 3,89 $\leq \frac{4}{3f+1} \implies$ f = 0,009425

For K = 32: $f = 0,00167$

The value of the maximum speed-up (value 3.89) for when K=4 and T=4 was calculated in the previous report and can be observed in Table 2.

In this case, we choose to not parallelize this functions, as the "while" cycle of the function kmeans needs to be sequential, as it garanties the corrected order of the actions. For example, organizing the last iterations before the first ones is not correct.

The creation of the points couldn't be parallelize as we need to ensure all points were correctly created for the correct execution of the program.

Memory Wall:

"The memory wall describes implications of the processor/memory performance gap that has grown steadily over the last several decades. If memory latency and bandwidth become insufficient to provide processors with enough instructions and data to continue computation, processors will effectively always be stalled waiting on memory. The trend of placing more and more cores on chip exacerbates the situation, since each core enjoys a relatively narrower channel

to shared memory resources. The problem is particularly acute in highly parallel systems, but occurs in platforms ranging from embedded systems to supercomputers, and is not limited to multiprocessors." - Encyclopedia of Parallel Computing by Professor Sally A. McKee & Dr. Robert W. Wisniewski

In our program we depend on shared memory between the threads, therefore all of them should know the cluster and the points they are working with.

As it is known, there is a memory/processor performance gap, so if our threads have a lot of misses, they are going to spend a lot of time waiting for data. The performance gap can be *"intensified"*, as there will be multiple threads sharing the *bus* which allows communication between CPU/memory. This can turn into a bottleneck as the number of threads increase.

We believe our previous program already presents some spatial locality, which will be verify important to try and reduce the impact of the memory wall.

Task Granularity and Parallelism Overhead

Analyzing the number of instructions of the sequential code and the number of instructions from the parallelized version, we can see that, in average, the maximum variation is of $0.17x10^9$ and the minimum is $0.02x10^9$, for K=4. When K=32, the maximum variation is $1.2x10^9$. Therefore we notice that there is a certain cost to manage the parallelization.

We can then consider looking into the task granularity and parallelism overhead of the implementation, as there could be some changes to be made.

Synchronization Overhead

On the algorithm that was initially designed we used the *pragma* directives of *reduction* and *barrier* to assure the synchronization between threads. This operations have a certain cost as they may increase the idle time and also have associated some computation (*reduction*). Unfortunately, in our algorithm these synchronization operations are necessary in order to get the result that is expected.

Compute time per parallel task

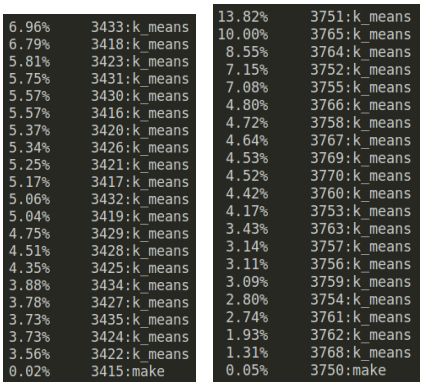

Fig. 1: Percentage of workload of each thread: Left: NThreads = 20 e K = 32 and Right: NThreads = 20 e K = 4

We can see in Fig 1 that when the number of threads is superior to the number of clusters, the work performed by each thread is not so balanced. Therefore, in a future correction we will try to improve the load balance of the threads.

Desirable Algorithm

After some research, we concluded that the best parallelization results are obtained when the algorithms consider spacial locality, load balancing, avoid idle time and data movements.

Spacial locality and data movements are what would help us reduce the impact of the memory wall and therefore allow us to take more advantage of the resources available. Considering the amount of idle time and trying to avoid it would not only help us continuously use the resources available but also reduce the impact that the percentage of sequential code had on our program. We should also aim to have a good load balance between threads.

IV. Number of Points Used

In the assignment, we were challenged to use input of different size adequate to different caches level. As we previously stated, we obtained our results in *SEARCH*.

In order to get the different search cache sizes (L1, L2 and L3) we consulted the size files on the folders index0, index1, index2 and index3 from the path */sys/devices/system/cpu/cpu0/cache/*. Since both index0 and index1 represent the L1 of cache memory, we have:

- L1 sizes = 64K each
- L2 size = 512K
- L3 size = 10236K

To get the RAM size we consulted the meminfo file from the folder */proc* (*command: cat /proc/meminfo ou vmstat -s*) : MemTotal size = 24522404 kB

The caches are divided in lines and for each miss, a line would be read into the cache. For example, when there is a miss in L1-cache, a line of this would be placed. The line size of all levels is 64 bytes.

The structure we used for the points and centroids use 12 bytes (two floats and one int) and 8 bytes (two floats), respectively. Therefore, we would be able to store 5.3 points and 8 centroids into a line of the cache.

We would also use three auxiliar arrays. Each would only store K floats/ints. Therefore, we can store up to 16 values of each array in a line, before we would have a miss.

We can conclude that, as the size of the L1-cache is 64kB and a size of a point is 12, we could fit 64K/12 = 5461 points. L2-cache could fit 43690 points, L3-cache can fit 873472. Lastly, the RAM can fit 2092578474 points

Leaving some space for the centroids, and the auxiliar arrays, we decided to use 5000, 40000, 800000 and 1600000 points in the different tests, execute in the next version of the algorithm with OpenMP. On the previous practical assignment we were testing our algorithm with 10 000 000 points and therefore more points than those that fit in L3. This might

mean that there will be misses in L1, L2 and L3 and that the majority of points will be loaded in memory.

For this pratical assignment, as previously mention, will be using the *SEARCH* machine, used during the course.

V. Code Versions

A. OpenMP

In a first attempt we tried to optimize the previously implemented OpenMP version, while taking into consideration the data calculated and the analysis made before.

For this purpose, we tried reducing the managing cost of parallel sections. For example, we noticed that in a single iteration of the algorithm, two different parallel sections were being initialized, contributing to the increase of runtime. Therefore, we made the alterations necessary, so only one parallelized section would be initialized in each iteration.

We used the **pragma directive simd** that when applied to a loop indicates the multiple iterations that can be executed concurrently. This helped us take advantage of vetorization which increases available memory bandwidth to cache, throughput of compute operations, is more power efficient and reduces frontend pressure (fewer instructions to decode).

In regards to the percentage of sequential code, as this was necessary to ensure the correct behaviour of the algorithm, we found that there wasn't a lot that could be done to reduce it.

By further analysing the spacial locality, there is a slight increase on the number of misses with the rise in number of threads. As previously mentioned, since memory wall has a great impact on efficiency, we tried to take advantage of the spacial locality.

Unlike what was stated in the previous assignment, different threads don't always have the same workload. If $N\_THREADS \leq K$, then the load if relatively balanced, otherwise this isn't ensured and some threads might have a higher workload than others. In this case, since the number of points is high, the function that assigns points might not be the main cause for the workload imbalance in our program. We believe this was caused by function *divisions_sums*. The **task pragma** is used to identify a block of code to be executed in parallel with the code outside the task region. We thought of using it to divide the function *divide_sums* function. This function is responsible for calculating the mean of the coordinates of all the centroids. This pragma directive allowed us to perform the different division of this function in parallel, as this function could be the cause of some load imbalance. When looking more attentively, this function would only be divided equally between threads when $K > NThreads$, therefore, our goal was to divide the workload more equally between threads, by dividing it. But unfortunately this didn't give us a better performance, as the cost of managing the tasks was not beneficial to the gain in load balance.

In order to get the correct result and avoid data races, in some cases, synchronization was necessary. As mentioned on the previous report, this was done resorting to pragma barrier and pragma reduction. On this practical assignment we trying to add pragma tasks we also added pragma taskwait to ensure the pragma task obtained the correct result.

Performance Tests

In order to analyse the impact that these changes had on our code we performed several tests.

The measurement of strong scaling is done by testing how the overall computational time of the job scales with the number of processing elements (being either threads or MPI processes), while the test for weak scaling is done by increasing both the job size and the number of processing elements.

For this reason when increasing the number of threads we can see we have some speed up however, this speed up in not close to the ideal speed-up presented in the graphs below. We can see that the number of instructions stay mostly the same.

We can also see that we was a significantly reductions is misses for when k=4. This is a result of a small change made to the code. Previously, we would add all the points of a cluster after they were assign. However, we now decided to do it simultaneously to the assignment of the points.

#T	TExec	SpeedUp	#Ix10^9	#CCx10^9	#Mx10^7	CPI
1	2,59	1,00	17,32	8,25	4,39	0,5
4	0,86	3,01	17,34	8,30	4,39	0,5
8	0,62	4,17	17,68	9,35	4,40	0,5
12	0,66	3,94	17,69	13,66	4,43	0,8
16	0,58	4,46	18,11	16,10	4,44	0,9
20	0,50	5,15	17,56	15,78	4,44	0,9

#T	TExec	SpeedUp	#Ix10^9	#CCx10^9	#Mx10^7	CPI
1	14,02	1,00	99,62	44,68	41,42	0,5
4	4,12	3,41	99,73	45,19	41,46	0,5
8	3,60	3,89	100,00	54,30	50,12	0,5
12	3,15	4,45	100,21	60,97	49,09	0,6
16	2,24	6,27	100,32	73,69	41,65	0,7
20	1,97	7,13	101,20	89,21	51,06	0,9

Tables 3 & 4: Results obtained using different numbers of threads with 4 and 32 clusters respectively

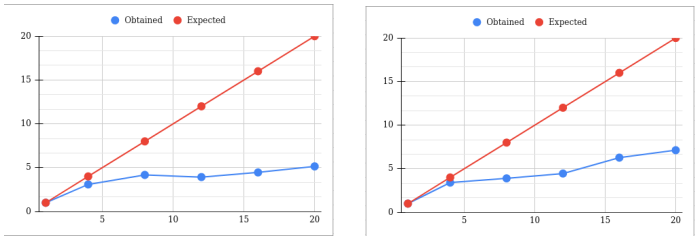

Fig 2 & 3: Speed-up graphic representation with 4 and 32 clusters

We decide to see how our code would react when the number of points it worked with changed. We used the number of points determined previously. As the number of points used would need to be storage in a lower cache level, we expected the missing values to increase as they did. However, we didn't expected the program to present a lower time of execution fo the higher number of points.

N	TExec	#Ix10^9	#CCx10^9	#Mx10^5
5000	0,46	7,29	23,80	7,23
40000	0,52	7,66	24,87	16,16
800000	0,60	8,58	24,19	34,20
1600000	0,21	4,45	7,88	75,61

Table 5: Results obtained using different numbers of points with 4 clusters

B. Message Passing Interface - MPI

Taking into consideration what is know about MPI, this is an algorithm that allows each processor to rapidly access its own memory without interference. This means that if each process has it's own private memory, then for programs that require them to access shared memory, MPI doesn't work well. For the k-means algorithm we then concluded that MPI would perform well.

C. Compute Unified Device Architecture - CUDA

Another approach was using CUDA. This parallel computing platform allows software to use certain types of graphics processing units (GPUs) for processing.

GPUs are more effective than central processing units (CPUs) for algorithms in situations where processing large blocks of data is done in parallel, which is what happens with k-means. CUDA exposes a fast shared memory region that can be shared among threads. This can be used as a user-managed cache, enabling higher bandwidth. Therefore, CUDA allowed us to work with a higher number of threads and, because of GPU, with shared memory.

For our CUDA implementation, we decided to keep a similar algorithm to the one used in the OpenMp solution, however, we used a *device* to execute the parallel parts of the implementation.

Therefore, we started the algorithm in a similar way to the previous implementation, where we sequentially generated the points that were going to be used in the host (CPU). Secondly, we allocated space for all the arrays we would need to store the necessary data in the device (GPU). These arrays included: an array for all points and one for centroids. Using arrays helped us calculate the mean. Afterwards we needed to copy all the values created in the CPU to the device, so this could be used by it.

As it was previously stated, the algorithm was quite similar to the one previously used. We assigned the points to their correct clusters and determined the new centroids iteratively, until all the necessary iterations to end the algorithm were performed.

CUDA allows the use of two different types of memory, global and shared, where Shared Memory is slightly faster then the Global. To take advantage of this, we decided to implement two different versions of the algorithm. One would only be using global memory and another would create a copy of the centroids in shared memory and use auxiliar arrays in shared to add the values of all the points allocated to each cluster.

We decided to used shared memory for addictions because, as we needed to use the atomic add to prevent possible Data Races, addictions would be slower operations. Therefore we hoped to make that process faster using shared memory.

Afterwards we collected some values to analyse. When testing both of these algorithms, for number of points N = 10000000, number of clusters K = 4, N_THREADS = 100 and N_BLOCKS = 100000, we obtained the following results:

Operations	Global Memory	Shared Memory
update_cluster_points	91.89%	76.86%
memcpy HtoD	5.67%	16.91%
initialize_sums_and_size	1.22%	3.11%
mean_sums	1.22%	3.11%
memcpy DtoH	0.00%	0.00%
Execution time	**Global Memory**	**Shared Memory**
Sequential	465088μs	547926μs
Parallel	575041μs	235649μs
Basic profilling	1040.75ms	784.212ms

Table 6: GPU usage and execution time using Global Memory vs Shared Memory algorithms

As we can see the function *update_cluster_points* uses a smaller percentage of GPU when using Shared Memory. We can then conclude that creating a copy of the centroids in shared memory and using auxiliar arrays in shared to add the values of all the points allocated to each cluster, is, as expected, faster and more efficient.

1) Weak Scaling: In order to analyse in detail our solution with CUDA, we tried testing it with weak scaling by changing the number of threads used proportionately to the number of points used.

For this analysis, we decided to choose a number of points that will fit in global memory and in L2-cache of GPU. We used the metrics from the Device *Kepler K20xm*. We considered that the size of L2-cache is 1572864 bytes and the size of the Global Memory is 5760MBytes (6039339008 bytes). Therefore, we could fit 131072 points in the L2-cache and 503278250 points in the global memory. Because of this, we decided to use 100000, 500000, 10000000 and 20000000.

Parameters	Operations	Global	Shared
N = 10000000;	**update clusters**	91.17%	76.80%
K = 4;	**memcpy HtoD**	6.41%	17.35%
Threads = 100;			
Blocks = 100000;	**Texec**	1019.69ms	800.248ms
N = 500000;	**update clusters**	91.59%	78.98%
K = 4;	**memcpy HtoD**	5.52%	13.59%
Threads = 100;			
Blocks = 5000;	**Texec**	53.6895ms	37.4502ms
N = 100000;	**update clusters**	92.14%	80.34%
K = 4;	**memcpy HtoD**	3.37%	8.38%
Threads = 100;			
Blocks = 1000;	**Texec**	13.407ms	9.72797ms
N = 100000;	**update clusters**	92.14%	80.34%
K = 4;	**memcpy HtoD**	3.37%	8.38%
Threads = 100;			
Blocks = 1000;	**Texec**	13.407ms	9.72797ms

Table 7: GPU usage and execution time using Global Memory vs Shared Memory algorithms

As predicted, even if the number of points increases a lot, the response capacity remains approximately the same. Therefore we can concluded that this algorithm presents a good scalability.

VI. Conclusion

After the analyse made we can observe that, despite obtaining a better execution time from the algorithm developed using OpenMP, we believe the algorithm developed with CUDA to be the most scalable one.

VII. Appendix

Material from the previous report:

Knowing that N was 10000000 and number of iterations is 20, because of an additional iteration to initialize the clusters, we present in table 2 the results of the calculated speed-up for different number of clusters, according to the calculations presented in the Appendix.

#T	1	4	8	12	16	20
K = 4	1	3,89	7,5	10,41	14	16,94
K = 32	1	3,98	7,91	11,8	15,64	19,44

Table 2: Expected Speed-up for different number of threads with 4 and 32 clusters

A. Sequential code

The sequential implementation has the following complexity: $O(N + K + N * K + It * (N * K + N + K + K))$.
The parallel implementation would have the complexity: $O(N + K + \frac{(N*K)}{\#Threads} + It * (\frac{(N*K)+N+K+K}{\#Threads}))$.

Our code's complexity can be expressed in pseudo-code with:

```
kmeas(){
    init();
    update_cluster_points();
    cycle:
        determine_new_centroid();
        update_cluster_points();
}
```

Each function's complexity is:

- init: $N + K$
- update_cluster_points: $N * K$
- determine_new_centroid: $N + K + K$