Parallel Patterns with C and OpenMP

Concurrency and Parallelism 2019/20

Ema Vieira, *50095* and André Atalaia, *51910*

Abstract—This project focused on implementing a parallel version of a set of parallel programming patterns: Map, Reduce, Scan, Pack, Gather, Scatter, Pipeline and Farm. These patterns were implemented by using the C programming language and the OpenMP API. Furthermore, this project required the development of scripts to run the tests, parse their output and generate plots, in order to automate the process of testing, collecting and processing data and visually analysing the results. Based on the evaluation of the performance of the parallel implementations under different conditions, such as the completion of heavier jobs and the increase of the quantity of work, we analyzed the results and suggested some further improvements.

Index Terms—Parallel Programming, Concurrency, Paralellism, Programming Patterns, OpenMP.

1 Introduction

In this project, we performed a comparative analysis on the performance of the sequential and parallel implementations of eight programming patterns. Although it may not be obvious at first sight, all computers are essentially parallel. This means that they are always running several jobs at the same time. Having this in mind and in order to make our parallel implementations scalable, we tried to split the work into several jobs and then run them at the same time.

In this report, we will give a brief description of the patterns followed by the explanation of our implementations. Next, we will describe the testing process, which variations we chose to test and how we collected and analysed the results. Finally, we will present the analysis of our results and some insights into how we could have improved our parallelization.

2 Implementation

In this section, we describe our implementation of the patterns. For the sake of simplicity, we will make use of the following equivalences:

- Src : the source array;
- N : number of elements in the source array;
- $Dest$: the target array.

The size of the $Dest$ array varies from pattern to pattern.

2.1 Map

The Map pattern applies a function to all positions of Src and stores their results in $Dest$. It is implemented by parallelizing the function call loop using `#pragma omp parallel for`, given that the function calls are all independent from each other.

2.2 Reduce

The Reduce pattern uses an associative and commutative combiner function to combine all the elements of Src into one single value stored in $Dest[0]$.

Our parallel implementation has two phases. The first phase is based on tiling, so we split the work into blocks of approximately the same size and then perform the reduction on each of them, separately [2]. In the second and last phase, we perform a global reduction over the results of the previously computed reductions. It is worth mentioning that, in this global reduction phase, we do not impose any order on the reductions, so functions like String concatenation might not produce the correct result because they are not commutative. It is also important to note that floating point addition and multiplication are approximately associative and might lead to precision errors.

2.3 Scan

The Scan pattern is similar to Reduce, but produces all partial reductions of Src and stores them in $Dest$. There are two variants of Scan, but the one we implemented was the inclusive one, which means that the each position $Dest[i]$ is a reduction of all positions of Src up to and including $Src[i]$ [2].

Our first approach was to implement the three-phase Scan, however we were having trouble with the implementation. Some colleagues reminded us of the tree-based Scan that was being lectured in the classes and we didn't finish the three-phase implementation.

After analysing the lecture slides on the Scan algorithm, we ended up implementing a recursive binary tree-based Scan [1]. One key aspect of this implementation is that it only works when N is a power of 2, due to the use of a binary tree. To allow all input sizes, our implementation of the Scan receives the Src array and, if N is not a power of 2, it expands the array to a size that matches the next power of 2 (e.g. If $N = 6$ then we expand Src to size 8).

2.4 Pack

The Pack pattern reads the elements from Src and moves only the chosen ones to contiguous memory, in our case to $Dest$ [2]. The elements written are chosen according to $filter$, an array of integers that contains 1 if the element is to be kept or 0 if the element is to be discarded.

Our implementation of Pack uses the Scan pattern, more specifically the prefix sum [1], in which we apply the Scan

with the addition operation to the $filter$ array. This produces an array of indexes, $bitsum$, that we use to know the position in which each element should be written. The final step in our algorithm makes use of the $filter$ to know which elements are to be kept, and if they are, it uses the $bitsum$ array as such: $Dest[bitsum[i] - 1] = Src[i]$.

2.5 Gather

The Gather pattern collects the elements of Src at the locations given by an array with indexes, $Src[filter[i]]$, and writes them in $Dest[i]$. This can be thought of as a combination of a Map with a random read [2] and, as such, it was implemented by making use of the `#pragma omp parallel for`.

2.6 Scatter

The Scatter pattern is similar to Gather but the $filter$ array contains write indexes instead of read indexes. In the end, each element $Dest[filter[i]]$ will have the value of $Src[i]$. However, when there are duplicates in the $filter$ array, it is not clear which value will be outputted because this can cause race conditions.

To solve this problem there are at least four different solutions [2]. We implemented an Atomic Scatter, which produces non-deterministic results by atomically writing the value read by the first thread to reach the code. Additionally, we decided to implement a version of Scatter with a simple `#pragma omp parallel for` that is to be used only with $filter$ arrays with no duplicates.

2.7 Pipeline

The Pipeline pattern receives a sequence of functions to apply, $workerList$, and the Src, and processes the work similarly to the processing in a factory. This pattern works by assigning a worker to execute the first function on a Src position and then the output from that function will be the input for the application of the second function to the same Src position, and so on. This is applied to all positions of Src in parallel.

Our implementation of the Pipeline patterns uses the matrix represented in Figure 1. Each position of the matrix represents the application of a function of $workerList$ to a position of the Src array. By iterating over the anti-diagonals of this matrix, we have no dependencies between the outputs and inputs of said functions. So, our implementation iterates over them and parallelizes the application of functions within each one [3]. One downside to this implementation is that, when the workers finish their jobs they have to wait for the last worker of the current anti-diagonal to finish before moving on to the next anti-diagonal. This could be improved by making the application of $workerList[i]$ to $Src[i]$ depend only on the application of $workerList[i-1]$ to the same Src position. We did not, however, have time to make this improvement in our project.

2.8 Farm

The Farm pattern is similar to the Map because it applies a function to all positions of Src and stores their results in $Dest$. However, the Farm pattern makes use of the Master-Slave model of parallelism, in which a Master thread is responsible only for distributing the jobs among the existing Slave threads. Upon completion of the jobs, the Slave threads report back to the Master and are assigned new jobs, if there are any left to do.

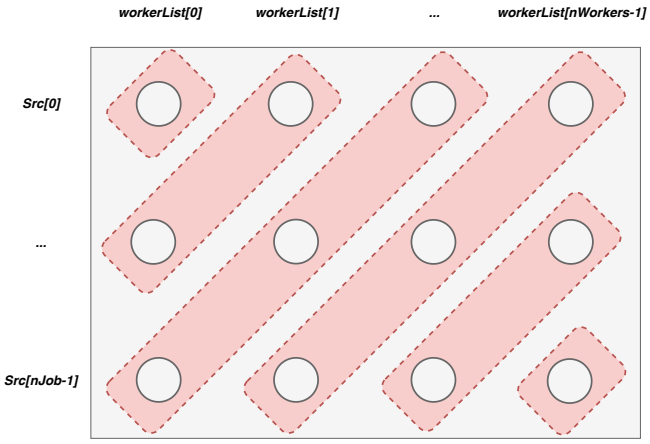

Fig. 1. Visual representation of the anti-diagonals of the parallel Pipeline implementation.

Our implementation uses an array of flags with the size of the number of workers, $flagWorkers$. This array is initialized with zeros, symbolizing that no worker has been assigned any job. Then, the Master thread proceeds to iterate over $flagWorkers$ and distribute jobs by using `#pragma omp task untied` and setting the corresponding flag to 1. After completion of the job, the workers set their own flag back to 0 as a way of communicating to the Master that they are ready to be assigned a new job. In order to prevent false sharing, we increased the size of $flagWorkers$ by adding padding of 60 bytes between each flag. This works well for processors with L1 cache lines of 64 bytes.

3 Evaluation and Analysis

All patterns were tested thoroughly. In order to guarantee this, we created several sets of tests and bash scripts to run them. After testing, we implemented python scripts that received the .cvs files outputted by the tests and plotted graphs. This allowed us to have a better visual understanding of the performance of our implementations.

We ran the tests on our personal machines as well as on a Linux node (node12) of the research cluster at DI-FCT-NOVA. This node has 4 AMD Opteron 6272 CPUs, each with 16 CPU Cores and 16 threads, totalling 64 hardware threads. The graphs below were plotted with the cluster results.

First, we ran the tests on our machines and took note of the results observed. Seen as we had limited time to run our tests on the cluster node, we carefully chose the tests we wanted to run. Some of the tests needed to be adapted because the run time on the cluster was usually double the time on our machines, so for example, when we ran tests with heavier jobs we resorted to making the size of the input array smaller, to reduce the run time.

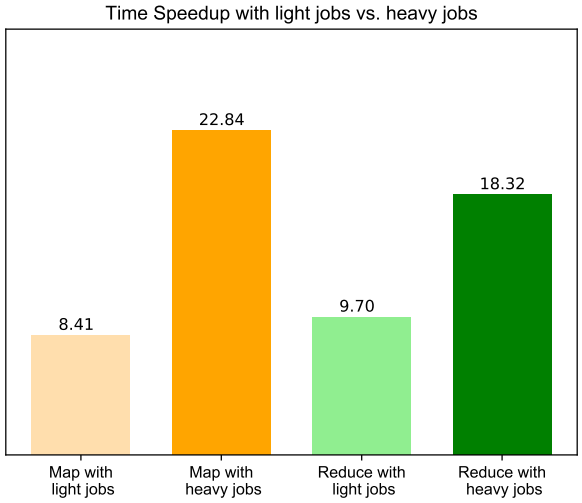

Fig. 2. Time Speedup of Map and Reduce with light vs. heavy jobs and 32 threads.

3.1 Types of Tests

We varied several parameters in order to fully test our patterns, namely:

- **Size of the work:** The amount of tasks to be completed, given by N and sometimes affected by the $filter$ array;
- **Granularity of the work:** The amount of work that is performed by a given task. We created heavier jobs that resemble the light jobs but execute a large empty loop;
- **Number of threads:** The number of available threads, set in `OMP_NUM_THREADS`;
- **Number of workers:** The total number of workers. This parameter only applies to the Pipeline and Farm patterns.

To reduce the bias created by run time outliers, all of our tests were run 3 times and the results were averaged.

3.2 Analysis of the results

3.2.1 Map

From the tests we ran on the Map pattern, it was clear that the parallel Map wields better results as we increase the size of the work. This is because the quantity of work makes up for the overhead of managing the threads. As is clear in Figure 2, when we increase the granularity of the work, the speedup also increases. This again, happens because if the complexity of the work increases, the overhead of managing the threads isn't as noticeable.

As expected, if the size of the work or its granularity was big enough, we obtained better results the more threads we added.

3.2.2 Reduce

The results obtained for the Reduce pattern were very similar to those of the Map, as seen in Figure 2. This pattern wielded better results as we increases the size of the work an its granularity. Again, if the previously mentioned parameters were big enough, the speedup increased with the number of available threads.

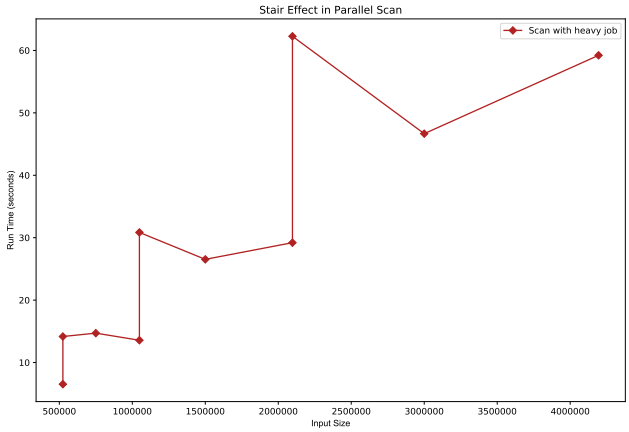

Fig. 3. Run time of Scan showing the stair effect.

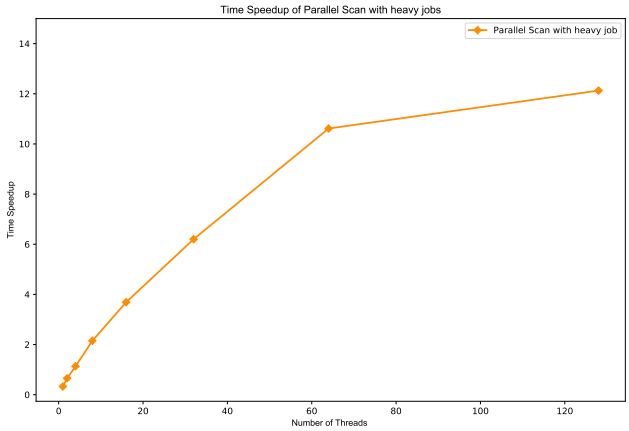

Fig. 4. Time Speedup of Scan with heavy jobs.

3.2.3 Scan

As explained in section 2.3, we implemented the Scan using a binary tree by expanding Src to the next power of 2. Therefore, the run time for this implementation presents a stair effect, as confirmed by Figure 3.

When analysing the results, it was clear that the performance increased as the granularity of the jobs increased, as shown in Figure 4, where we can see the speedup obtained with heavier jobs. However, when we tested our implementation with lighter jobs there was a significant slow down, justified by the overhead introduced by the management of the threads and the creation of the tree. To reduce this overhead, we suggest studying the introduction of a sequential cutoff. However, we did not have time to test its benefits in our project.

3.2.4 Pack

As we mentioned before, the Pack pattern uses the Scan pattern with the addition operation to calculate the prefix sum. This approximates its behaviour to that of the Scan with lighter jobs and, because of that, it didn't wield better results than its sequential version.

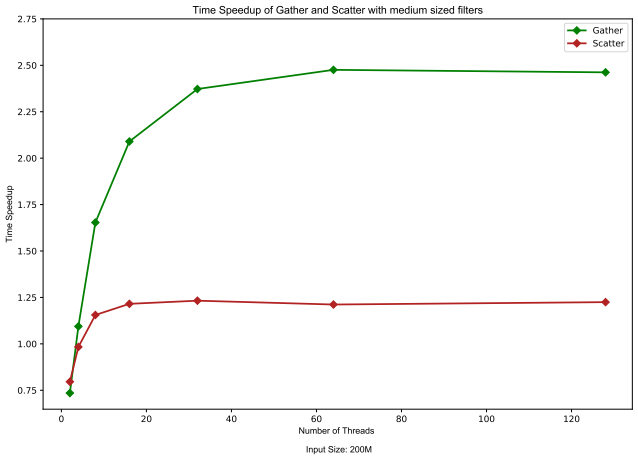

Fig. 5. Time Speedup of Gather and Scatter with medium sized filters.

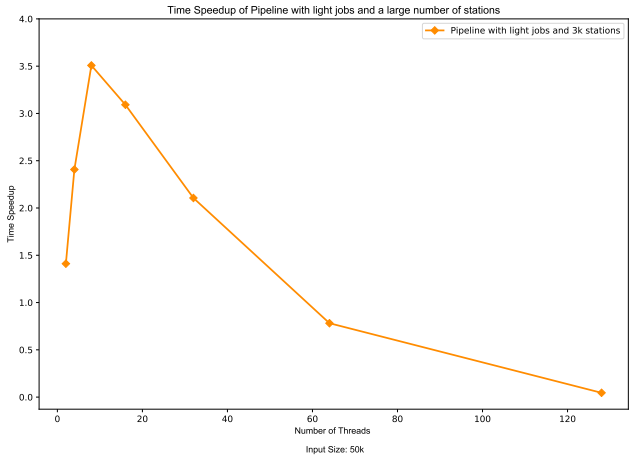

Fig. 6. Time Speedup of Pipeline with light jobs and 3k stations.

3.2.5 *Gather*

The results obtained for the Gather pattern were very similar to those of the Map pattern with lighter jobs. Gather wielded better results as we increases the size of the work and it also proved beneficial to increase the number of available threads, as can be seen in Figure 5. Obviously, if we increase the size of the $filter$ array, the run time will also increase.

3.2.6 *Scatter*

The non atomic Scatter pattern presents only a slight speedup, as seen in Figure 5. This is because there might be threads competing to write in the same position and causing slow downs. We chose to include this version because we obtained poor run times with the Atomic Scatter, and we believe that the benefit of allowing duplicates in the $filter$ array is not worth the heavy loss in performance.

3.2.7 *Pipeline*

The Pipeline pattern, when tested with lighter jobs, proved efficient only when the number of work stations was significantly high. This is because the size of the anti-diagonals and amount of work make up for the overhead of managing the threads. However, when the number of threads increases, so does the overhead and the performance starts declining, as we can see in Figure 6. As mentioned in section 2.7, the threads wait for the parallelization of the current anti-diagonal to finish before moving on to the next anti-diagonal. So, when tested with heavier jobs, instead of showing improvements, our implementation closely resembles the sequential version, because the threads might have to wait for a significant amount of time for the last thread to conclude such a heavy job.

It is important to mention that, since our implementation heavily depends on the size of the largest anti-diagonals - $min(N, nWorkers)$ - it is not worth adding more threads than that size.

3.2.8 *Farm*

When we tested the Farm pattern with lighter jobs it did not present any speedups, because the granularity of the jobs did not make up for the overhead of managing the threads. However, when testing with heavier jobs, the speedups were very good, as we can see in Figure 7. From this Figure it is also clear that, when the number of threads surpasses the number of workers, some of the threads are idle and the overhead of managing them takes a toll on the speedup.

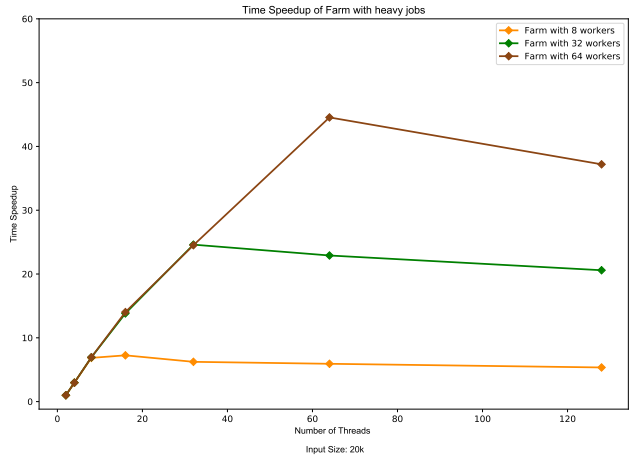

Fig. 7. Time Speedup of Farm with heavy jobs and varying number of workers.

4 Conclusion

We believe that this project gave us a new mind set when it comes to testing our work because, after finishing the project, some downfalls of our implementations were not clear to us. However, after extensively testing our code, we came to several realizations and were able to better understand the problems with our implementations. With this said, we would have liked to have more time to correct these problems and test our new solutions.

Furthermore, this project made it clear to us that automating every aspect of testing, collecting data and analysing it is much more efficient than doing it manually and heavily prevents human error.

Additional Work

In addition to the implementation of the patterns, we also implemented **Validation Tests**, to guarantee that the output of the parallel version coincided with that of the sequential version. After validating our output, we implemented several **Unit Testing functions** with varying parameters, such as job granularity and filter size. To run these tests, we developed **bash scripts** that ran our code while varying the number of threads and the input size. After the testing process, we created **python scripts** that read the .cvs output files, plotted graphs and helped us have a better visual understanding of our results.

Workload Distribution

The distribution of the workload during this project was split evenly between the group members (50-50). Both parts dedicated their time and put a lot of effort into delivering the final product.

While developing the project, even though we used Git Workflow and committed individual work, we were always connected via voice call and screen share, helping each other and following a kind of "Distance Peer Programming".

Acknowledgments

We would like to thank the members of **group 27**: João Monteiro (50576), for suggesting we implement the Scan pattern with a binary tree and for later helping us improve its the spacial complexity; and Luís Rosário (50547), for bringing to our attention that there could be more workers than threads in the Farm pattern.

References

[1] Lourenço, J., 2020. *Parallel Algorithms*. Lecture slides. Concurrency and Parallelism, FCT-UNL. Lecture delivered on March 31st 2020.

[2] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. https://www.elsevier.com/books/structured-parallel-programming/mccool/978-0-12-415993-8.

[3] Solihin, Y., 2009. *Fundamentals of Parallel Computer Architecture: Multichip and Multicore Systems*. Solihin Publishing & Consulting LLC. https://books.google.pt/books?id=jPHzPgAACAAJ.