Concurrency and Parallelism
Cilk$^+$ Parallel Patterns Implementation

André Rosa
48043
af.rosa@campus.fct.unl.pt

João Geraldo
49543
j.geraldo@campus.fct.unl.pt

Rúben Silva
47134
rfc.silva@campus.fct.unl.pt

***Abstract*—In recent years, the parallel computation paradigm has been emerging, as a consequence to: the huge growth on the amount of data that needs to be processed and analyzed, and to the switching of processors' architecture evolution process from becoming smaller and having faster clock speeds, to integrate parallel functionalities, such as multiple cores, hardware threads and vector operations. Therefore, it is imperative to build algorithms that explore these functionalities to increase the efficiency on processing such huge amounts of data. However, building algorithms on top of these functionalities can sometimes be a difficult and complex task, due to them being low level primitives and sometimes platform dependent. Thus, parallel functions and libraries can provide an easy and generic way to better utilize these resources, when available. Therefore, we developed a library, that implements some of the most well known parallel patterns, and that can be easily integrated in any already existing sequential program, to improve its efficiency. We developed each algorithm to try to achieve the maximum possible parallel slack and ...**

In order to be able to use it in a vast amount of use cases, these implementations are independent both from the data types they are manipulating and the parallel functionalities provided by each specific hardware platform.

What was our approach? What were the results? What did you learn?

Finally, we conducted a preliminary experimental evaluation on the performance of the different implemented alternatives, comparing them with their corresponding sequential version, that showed ... .

***Index Terms*—Parallel Algorithms, Cilk$^+$**

1. Introduction

Nowadays, the parallel paradigm is a matter of great importance, and the demand for more scalable and efficient data processing algorithms has increased.

In the one hand, the amount of data produced every day grows exponentially [meter uma citação], requiring more computational power to process it at the same speeds.

On the other hand, due to single-core processors are reaching the physical limits of improvement through shrinking its components and increasing its clock speed, a revolution happened in the processor's architecture design. Processors evolution switched to have parallel functionalities, such as multiple cores and hardware threads as well as vector operations, that allow the simultaneously execution of processes and parallel manipulation of multiple data [1].

For these reasons, it's crucial to take advantage of these features to build algorithms that allow the processing of such huge amounts of data, having good levels of scalability and latency. However, build such algorithms can be a very difficult and complex task, not only because they have to make use of the previous low level parallel primitives, but also because those primitives can be platform dependent and are highly heterogeneous, e.g., not all the processors provide vector operations and hardware threads and the amount of cores may vary greatly, even within the same manufacture, take Intel as an example - Intel's i7 lineup has chips with only 2 cores up to chips with a whopping 16 cores.

Just like sequential algorithms, the parallel ones also have a set of specific recurring configuration of computations and data accesses. This configurations are called parallel patterns [2] and they can help build parallel algorithms, serving as building blocs that can be glued together to produce an algorithm. Thus, a parallel algorithm can be seen as a composition of these parallel patterns.

On that account, we designed and implemented a library that contains some of the most well known parallel patterns.

This library provides an easy way to build parallel programs as well as to update already-built sequential programs to make use of the hardware parallel functionalities, allowing the application-level developer to integrate such patterns and give its application a boost of performance with little effort. We resorted to Intel's Cilk$^+$ [3] to implement those patterns. In order to be able to use it in a vast amount of use cases, these implementations are independent both from the data types they are manipulating and the parallel functionalities provided by each specific hardware platform. We developed each algorithm to try to achieve the maximum possible parallel slack and ... [mudar um bocado isto]

The remaining of this report is structured as follows: at Section 2 we present the architecture of algorithm and at Section 3 we discuss their relevant implementation details. The settings of the experiments along with their results are presented in Section 4. Lastly, we conclude this report with the conclusions, at Section 5.

2. Architecture

In this Section, we describe the architecture of each of the implement patterns: Map, Reduce, Scan, Pack, Split, Gather, Scatter, Pipeline and Farm. [citações para os patterns]

2.1. Map

The map pattern represents the independent application of a function to every element of a collection, and thus each operation can be executed in parallel. This function needs to be pure, e.g., not have side effects in order to be parallelizable. However, the computational weight of the aplication of the function might be too small in comparison to the overhead of parallelizing the tasks, leading to using the sequential version being a better option. Thus, to overcome this problem, the elements can be grouped in batches, that are process in parallel, and, within each batch, the elements are processed sequentially. The size of each batch is called *Grain Size*. [Dizer mais o que?]

2.2. Reduce

The reduce pattern represents the application of a pairwise associative operation to all the elements of a collection, producing a single element. Since the operation is associate, multiple applications (to different elements) can be done in parallel. The execution of the parallel version of these pattern produces a binary tree, where the child nodes are the operands and the parent is the result of the operation.

In our library, we implemented two versions of reduce: (regular) reduce and tiled reduce. In regular reduce, the elements of the collection are paired and the worker (operator) is applied in parallel to each pair. This process is repeated to their results until there is only one element, the reduce result. However, the weight of the application of the operators may be so small compared to the parallel overhead, that it does not pay to use the parallel version. Thus, we decided to implement an alternative version for the reduce - tiled reduce, which groups several elements of the collection in a tile, instead of only two, and executes the sequential version for each in parallel.

problem of semi - associative operations such as floating point arithmetic.

2.3. Scan

The Scan pattern is very similar to the Reduce pattern with the difference that the output is a sequence of the results of all the intermediate operations and not just the last result. This sequence is ordered in the sense that the element in position i of the results from applying the operator to the element in position i-1 with the element of position i of the input collection. We tried to develop our own implementation of scan, however, we decided to use this [Citação], because it is better. [mudar este discurso]

Briefly, this algorithm first builds a tree with bottom-up approach, similar to reduce, where the results of the intermediate operations are computed pairwise. Then, it traverses the tree top-down, passing to the left child the value inherited from its parent (the neutral element of the operation in the case of the root) and to the right child the result of applying the operator to the value inherited with the result of the intermediate computation of its left brother. In it reaches the leafs, the value to insert in the output collection corresponds to applying the operator to the inherited value and its own intermediate value.

[falar da complexidade disto e que tem mais work para ter menos span]

[Está um bocado mal explicado]

2.4. Pack

The Pack pattern is used to eliminate wasted space in a sparse collection and to handle variable-rate output from a map. From within map, each function activation is allowed to either keep or discard its outputs. The survivors are then packed together into a single collection. We consider make our own algorithm to solve pack but because it was not as efficient as the algorithm proposed by [those dudes] so we opted to implement that algorithm.

2.5. Split

The Split[1] pattern is a variant of pack which does not discard elements, but instead packs them to the top or bottom of the output collection. In the split pattern, the order of the elements within each output segment is the same as their relative position in the input (in other words, it is a "stable" reordering), and the total output size is always exactly the same size as the input. Split can obviously be implemented by running pack twice and merging the results. But because this would't be the most efficient implementation we developed a more direct algorithm to improves latency and thus more time efficient.

2.6. Gather

The Gather pattern takes a collection and a sequence of indices in the previous collection and produces a collection composed by the elements of the input collection that correspond to the index on that position. Since only multiple reads can happen on the same element simultaneously, then each element of the index sequence can be processed in parallel. We implemented this pattern resorting to the Map pattern, where the worker function corresponds to reading the element at the given index from the input collection and writing it in the corresponding slot in the sequence.

[Está um bocado mal explicado]

1. Optional Extra Pattern

2.7. Scatter

The Scatter pattern takes a collection and a sequence of indices in the previous collection and puts the element of the position i at index[i]. There is a problem with this pattern that is if the index collection has repeated elements then dataraces occur. In our implementation, we leaved that verification to the user of the pattern because it would add too much weight to the operation and we think it should be a concern of the program that uses the function and not the function itself (pre-condition). We implemented this pattern resorting to map, where the worker copies the value in the position i of the input array to position index[i] in the output array.

[Está um bocado mal explicado]

2.8. Pipeline

The Pipeline pattern represents the sequential application of multiple operators (or stages) at the same element. Due to having multiple operators, they can be applied in parallel at different elements. However, having only one element at each stage can be too slow and thus, we implemented an alternative parallel version of pipeline - pipeline farm, that have multiple elements being processed at each stage in parallel and thus increase the throughtput of the algorithm. This alternative version results from the combination of the Pipeline and Farm parallel patterns.

2.9. Farm

The Farm pattern is very similar to Map with the constraint that the maximum amount of parallel operations is limited to the number of farms (parameter of the pattern). Thus, each job can be distributed evenly through all the farms and then processed sequentially in the respective farm.

3. Implementation

In this Section, we present some implementation details of our algorithms, that we found relevant.

In the Map pattern implementation we would like to point out that we delegated the splitting of the source array into batches to the Cilk$^+$ runtime. We tried to make the batching separation ourselfs, however, because we were getting poorer outcomes, and because we couldn't make assumptions on data types we were manipulating, and thus not know its size, we decided make use of Cilk$^+$'s automatic splitting.

In the Reduce pattern implementation, to avoid dataraces when trying to keep the intermediates results contiguous in memory (for faster sequential processing), we decided to use two intermediate arrays, one to read and another to write, that switch roles at each iteration (level on the execution tree) of reduce. This allowed to not keep all the intermediate results in memory, because they are only needed to compute the next level results, and at the same time improve the processing of each level by keeping all the elements contiguous and thus, increase the processing of the sequential version algorithm on each batch (or pair of elements in the case of regular reduce).

4. Experimental Evaluation

In this Section we present an experimental evaluation of the implemented parallel algorithms, comparing their performance against their sequential version.

4.1. Experimental Setting

To evaluate the performance of our parallel algorithms, we built a tester application that, for each algorithm, runs each of its versions: sequential, parallel and alternative, when there is one; and measure its latency, e.g., the elapsed time between the start and finish of the algorithm's version. For a given algorithm, it computes the latency for a sequence of job sizes, allowing us to measure the behavior of our solution regarding the input size. Each algorithm version for a given job amount, is run a parameterizable amount of times, in order to compute the average of their results and thus smooth the noise introduced by other processes and hardware details. The precision of the time measures are in the magnitude of microseconds (us), but the results are presented in seconds to improve readability. Our experimental evaluation was executed in node9, a computer with 16 processors: Dual-Core AMD Opteron 8220 with 1000 MHz, Cache of 1024 Kb and TLB size of 1024 4K pages; and 28.3 GB of RAM. The results of these experiments are presented in the following section.

4.2. Experimental Results

The Figure 1 has the results obtained, for each algorithm, on the experiments.

Análise comparatória da performance dizendo porque correu bem e porque correu mal para cada um. Análise geral, comparando a performance dos vaŕios uns cons os outros. Ex o map é o que aprenseta melhores resultados comparativamente à versão sequencial, bla bla Podiamos medir a difenreça entre a paralela e a sequencial dividdo e assim tinhamos um valor que ea comparável com os outro salgoritmos

5. Conclusion

The conclusion goes here.

Acknowledgments

The authors would like to thank...

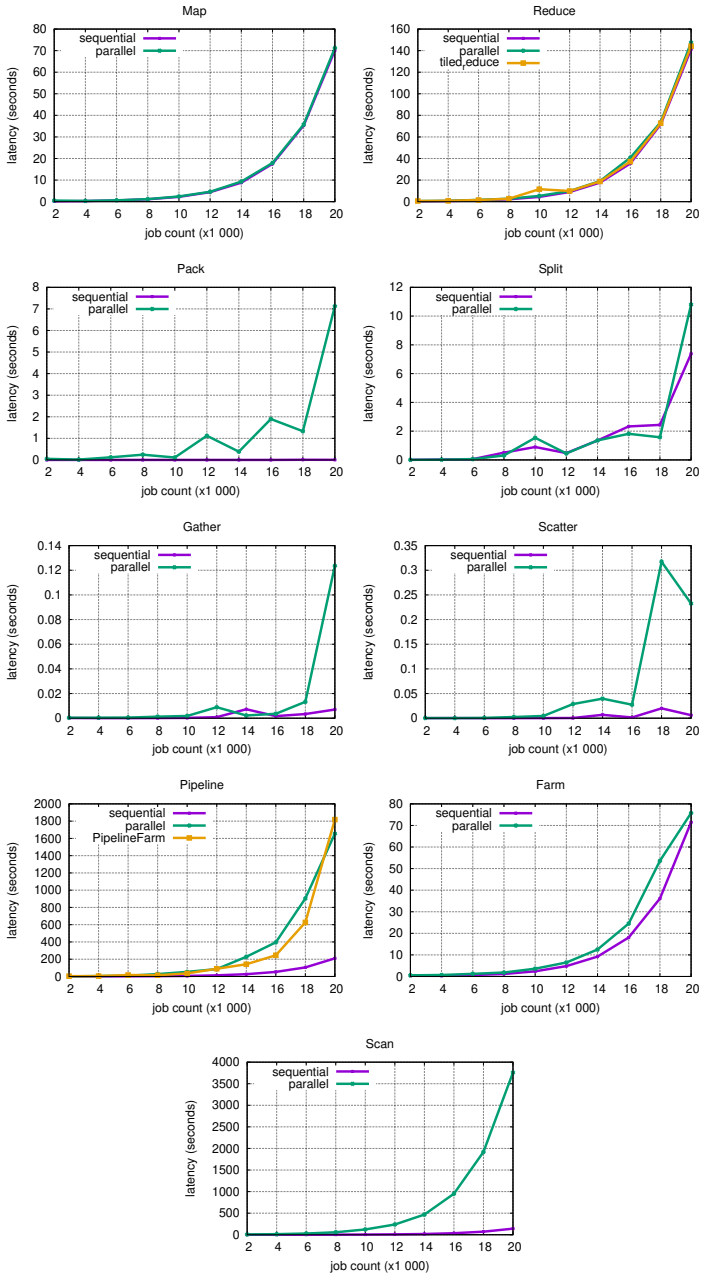

Figure 1. Latency of the algorithms.

Comments

References

[1] A. Fasiku, O. Oyinloye, S. Falaki, and O. Adewale, "Performance evaluation of multicore processors," *International Journal of Engineering and Technology*, vol. 4, no. 1, 2014.

[2] M. D. McCool, "Structured parallel programming with deterministic patterns," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. USENIX Association, 2010, pp. 5–5.

[3] A. D. Robison, "Cilk plus: Language support for thread and vector parallelism," *Talk at HP-CAST*, vol. 18, p. 25, 2012.