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*—Parallel computing is a type of computation in which several tasks are executed simultaneously. In recent years, the parallel computation paradigm has grown in relevance as a consequence of two modern trends: the explosive increase in the amount of data available to process and analyze; and the shift in processor development from smaller, faster single-core processors, to processors with multiple cores, hardware threads and vector operations support. In order to fully utilize the potential of parallel hardware and more efficiently process large amounts of data, it is imperative that algorithms are designed with parallelism in mind. The conception of parallel algorithms is made difficult by the very complex nature of parallel computation and by the fact that parallel hardware functionalites exist as low level primitives and are sometimes platform dependent. Parallel functions and libraries provide an easy and generic way to better utilize parallel hardware resources, when available. In this context, we developed a library that implements some of the most well known parallel patterns, and can be easily integrated into any already existing sequential program to improve its efficiency.**

In the implementation of each parallel pattern, we sought to maximize the amount of parallel slack i.e. excess of potential parallelism versus actual amount of hardware parallelism. This approach guarantees an efficient use of hardware resources, since different machines will be able to execute the parallel patterns with varying amounts of parallelism. We prioritized the decrease in execution time when implementing the various patterns, which for some of them meant increasing the amount of reserved memory to ensure better data locality and cache usage. Certain patterns were implemented in a generic, parameterizable way, which allows for the empirical finding of the set of parameters that result in the most efficient execution. In order to be able to use the library in a vast amount of use cases, the pattern implementations are independent both from the data types they are manipulating and the parallel functionalities provided by each specific hardware platform.

A testing tool was created to evaluate the performance of the implemented parallel patterns and compare it to the performance of their sequential counterparts. Making use of this testing tool, we conducted a preliminary experimental evaluation, which showed noticeable gains in performance for several of the parallel patterns versus their sequential alternatives. In this report, we discuss the results obtained from the performace evaluation and present possible explanations for the differences in computational performance for the various patterns.

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

***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 at impressive rates [1], requiring more computational power to process it at useful rates.

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 simultaneous execution of processes and parallel manipulation of multiple data. 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, i.e., not all the processors provide vector operations and hardware threads and the number of cores may vary greatly, even within the same manufacturer, 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 a specific recurring configuration of computations and data accesses. These configurations are called parallel patterns [2] and they can help build parallel algorithms, serving as building blocks 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 in our library: Map, Reduce, Scan, Pack, Split, Gather, Scatter, Pipeline and Farm. [citações para os patterns]

2.1. Map

The Map pattern consists on the independent application of the same function (or worker) to every element of a collection. This function needs to be pure, i.e., not have side effects in order to be parallelizable. As the applications of the function are independent from one another, they can be executed in parallel. However, sometimes the computational weight of the function might be too small in comparison to the overhead of parallelizing their execution, leading to worst results than utilizing the sequential version. Thus, to overcome this problem, the elements of the collection can be grouped in batches, which are processed in parallel, and, within each batch, their elements are processed sequentially. The size of each batch is called *Grain Size* and the separation in groups is called *Granularity*. We implement our map following that behavior.

2.2. Reduce

The Reduce pattern consists on applying a pairwise associative operation (e.g. Add, Multiply) to all the elements of a collection, producing a single element. Since the operation is associative, 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, being that the result of reduce corresponds to the root of this tree.

In our library, we decided to 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 operator may be too small compared to the parallelization overhead, that it does not pay to use the parallel version. Thus, we decided to implement the alternative version, tiled reduce, which groups several elements of the collection in a tile, instead of only two. This tiles are then processed in parallel, being that for each one, a sequential version of reduce is executed.

[Secalhar explicar mais em detalhe a nossa implementação]

2.3. Scan

The Scan pattern is very similar to the Reduce pattern with the difference that the output the sequence of results of all the intermediate operations and not just the last result. The order of the sequence must be the same as the order of the sequential version.

We decide to implement this version of Scan [citar].

Briefly, this algorithm first builds a tree with a 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 leaves, 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 consists on removing elements from a collection, keeping the remaining elements at contiguous positions, i.e., without unused space between them. The elements to remove (or to keep) are identified through a binary mask or a filter function function. In this pattern, the order of the elements is the same as their relative position in the input.

This pattern is generally used to eliminate wasted space in a sparse collection and to handle variable-rate output from a map (or other patterns), where the worker is allowed to either keep or discard its outputs. We considered designing our own algorithm to implement Pack, however, the version proposed in the classes was more efficient, and thus we opted to implement that instead. That version resorts to Scan to compute the sum of the mask elements, originating the bitsum. That sequence, gives, for each true element on the mask, the corresponding input's element position in the

output collection. Then, a map is applied to the bitsum with a worker that for each true element of the mask, inserts the corresponding element of the input collection into the output collection, in the position defined by the corresponding bitsum.

This implementation have work O(n) and span O(log n). [melhorar isto]

2.5. Split

The Split[1] pattern is a variant of pack which does not discard elements and instead packs them after the last element of the elements whose mask value was true, i.e., splits the true elements to one side of the collection and the falses to the other. In this pattern, similarly to pack, the the order of the elements, within each output segment, is the same as their relative position in the input, in other words, it's a "stable" reordering, being that output size is always exactly the same size as the input's. Split can obviously be implemented by running pack twice and then merging the results. However, because this wouldn't be the most efficient implementation, we developed a more direct algorithm that improves latency. First, it computes the inverse of the mask, turning the falses into trues and vice-versa. Then, parallel executes two scans, one on the original mask and another on the inverted mask, similarly to pack, producing positiveBitsum and negativeBitsum collections respectively. Afterwards, waits for the termination of the scans and then applies a single map across the input collection, with a worker that, for each element, if it's true on the original mask, insert it in the output collection, at the position given by positiveBitsum, else insert it in the output collection, at the position given by negativeBitsum plus an offset that corresponds to the amount of elements that are true in the mask and this value is given by the value on the last element of the positiveBitsum.

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]

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 data races occur. In our implementation, we left 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]

1. Optional Extra Pattern

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 to 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 throughput 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 ourselves, 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 data-races 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, i.e., 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 27 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

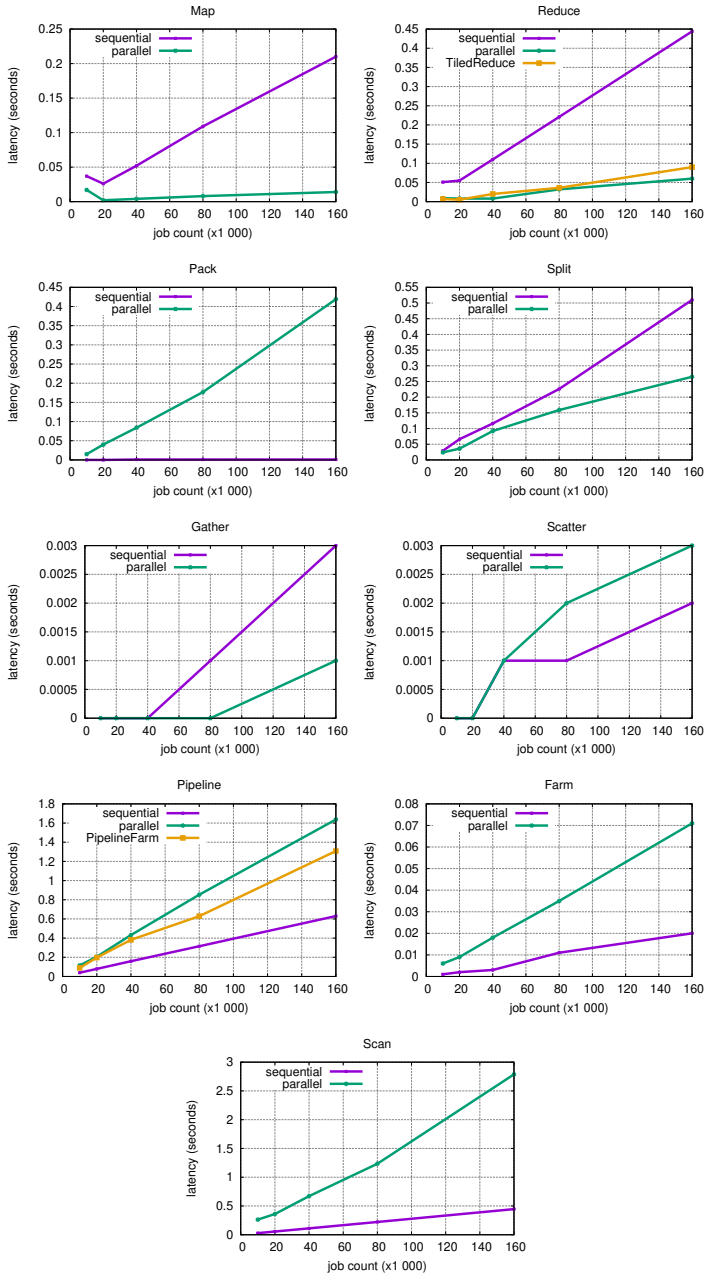

Figure 1. Latency of the algorithms.

5. Conclusion

The conclusion goes here.

Acknowledgments

The authors would like to thank...

Comments

References

[1] A. McAfee, E. Brynjolfsson, T. H. Davenport, D. Patil, and D. Barton, "Big data: the management revolution," *Harvard business review*, vol. 90, no. 10, pp. 60–68, 2012.

[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.