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 emerged as a consequence to the explosive increase in the amount of data available to process and the shift in processor's architecture evolution to integrate parallel functionalities, such as multiple cores, hardware threads and vector operations. Therefore, in order to fully utilize the potential of parallel hardware and more efficiently process large amounts of data, it is imperative to design algorithms that explore these functionalities.**

However, designing algorithms on top of them can sometimes be a difficult and complex task, due to very complex nature of parallel computations and by the fact that parallel hardware functionalities exist as low level primitives, being 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, allowing the composition of algorithms and, at the same time, can easily be integrated with any already existing sequential program, to improve its efficiency. 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.

In the implementation of each parallel pattern, we sought to maximize the amount of parallel slack. We also 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 flexibility in the distribution of work regarding the computational weight of the operations and the size of the data types being manipulated, resulting in the most efficient execution.

Finally, we conducted a preliminary experimental evaluation on the performance of the different implemented alternatives, comparing them with their corresponding sequential counterparts, that showed noticeable gains in performance for several of the parallel patterns.

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

1. Introduction

Parallel computing is a type of computation in which several independent tasks are executed simultaneously. In recent years, the parallel computation paradigm has grown in relevance as a consequence of two modern trends: the amount of data to process and the architectural evolution of processors. In the one hand, the explosive increase in the amount of data produced every day [2], lead to a demand of more scalabel and efficient data processing algorithms. On the other hand, due to the fact that single-core processors are reaching the physical limits of improvement through increasing its clock frequency and shrinking of its components, to decrease the communication time between them, a revolution happened in the processor's architecture design. Consequently, processors' architecture evolution switched to insert parallel functionalities, such as having multiple cores and hardware threads as well as supporting vector operations, that allow the simultaneous execution of processes and parallel manipulation of multiple data.

For these reasons, it is imperative that algorithms are designed with parallelism in mind, in order to fully take advantage of the potential of the parallel hardware they are being executed on, to process large amounts of data more efficiently, and at the same time provide scalability, to be prepared to the evolution of both the hardware and the amount of data produced.

However, the conception of parallel algorithms can be a very difficult and complex task due to the very complex nature of parallel computations, and by the fact that parallel hardware functionalities exist as low level primitives, being sometimes platform dependent and 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. Parallel functions and libraries provide an easy and generic way to better utilize parallel hardware resources, when available.

Simultaneously, 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 [3] and they can help conceive

parallel algorithms, serving as building blocks that can be glued together to produce an algorithm.

On that account, we designed and implemented a library that contains some of the most well known parallel patterns, providing at the same time abstraction from the available hardware resources. To be able to use it in a vast amount of use cases, these implementations were also designed to be independent from the data types they are manipulating. Thus, this library provides an easy way to construct parallel programs as well as integrate it with little effort on already-built sequential programs, making use of the available hardware parallel functionalities and consequently improve its efficiency. To implement those patterns, we resorted to Intel's Cilk$^+$ [4], a C/C++ extension that support data and task parallelism. 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 of the implemented 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 flexibility in the distribution of work regarding the computational weight of the operations and the size of the data types being manipulated, resulting in the most efficient execution.

The remaining of this report is structured as follows: at Section 2 we present the architecture of the algorithms and at Section 3 we discuss their relevant implementation details. The settings of the experimental evaluation, along with a discussion about the results obtained and present possible explanations for the differences in computational performance for the various patterns, that 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.

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.

2.3. Scan

The Scan pattern is very similar to the Reduce pattern, with the difference that it returns a sequence, with the results of all the intermediate operations, and not just the result of the last operation. The order of the sequence must be the same as the order of the sequential version's result.

We decided to implement this version of Scan [1], that is composed by two phases. In the first, it builds a tree, in a bottom-up approach, similarly to reduce, where the nodes are the results of the intermediate operations are computed pairwise, with the exception of the leafs, that correspond to input elements. The root of these tree corresponds to the result of applying the operator to all the input's elements, i.e., it's the same as the reduce value. In the second phase, it traverses the tree, in a top-down approach, passing to the left child of each node, 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. One important aspect of this algorithm is that the total work to be done is higher, having complexity O(log n), however the the value of the span decreases, to O(log n).

This implementation has work O(n) but its span is only O (log n).

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.

Due to resorting to Scan and the fact that Map has work of O(n) and span of O(1), this implementation also have work O(n) and span O(log n).

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 false 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 false 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.

1. Optional Extra Pattern

2.6. Gather

The Gather pattern takes a input collection and a sequence of indexes, producing a output collection, that for each element in the index sequence, it inserts at that position, the value of the input collection that is at that index value. Thus, the size of the output collection is exactly the same as the size of the index sequence.

Since only multiple reads can happen on the same element simultaneously (no data races), each element of the index sequence can be processed in parallel. Therefore, we implemented this pattern resorting to the Map pattern, where the worker function reads the element at the given index from the input collection and writes it in the corresponding position on the output sequence.

2.7. Scatter

The Scatter pattern is the inverse of Gather. It also takes an input collection and a sequence of indexes. However, instead of reading the corresponding input elements referenced by the indexes, it writes the input value at that index, in the output sequence. Consequently, there is a problem with this pattern when there are repeated elements in the index sequence, leading to data races. 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 also implemented this pattern resorting to Map, where the worker copies the value in the position i of the input array to position contained at the position i of the index array, to the output array.

2.8. Pipeline

The Pipeline pattern represents the sequential application of multiple operators (or stages) at the same element, being very similar to an assembly line. Due to having multiple operators, they can be applied in parallel to different elements. However, having only one element at each stage can be a bottle neck to the performance and thus, we decided to implement an alternative parallel version of pipeline - pipeline farm, that results from the combination of the Pipeline and Farm parallel patterns. Therefore, there are multiple elements being processed, in parallel, at each stage of the pipeline, consequently increasing the throughput of the algorithm.

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). Consequently, we distributed each job evenly through all the farms, that process sequentially their attributed input elements.

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 milliseconds (ms), but the results are presented in seconds to improve readability.

For each algorithm, we utilized computational heavy workers, that do multiple arithmetic computations, allowing to distinguish better the performance of the different versions.

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

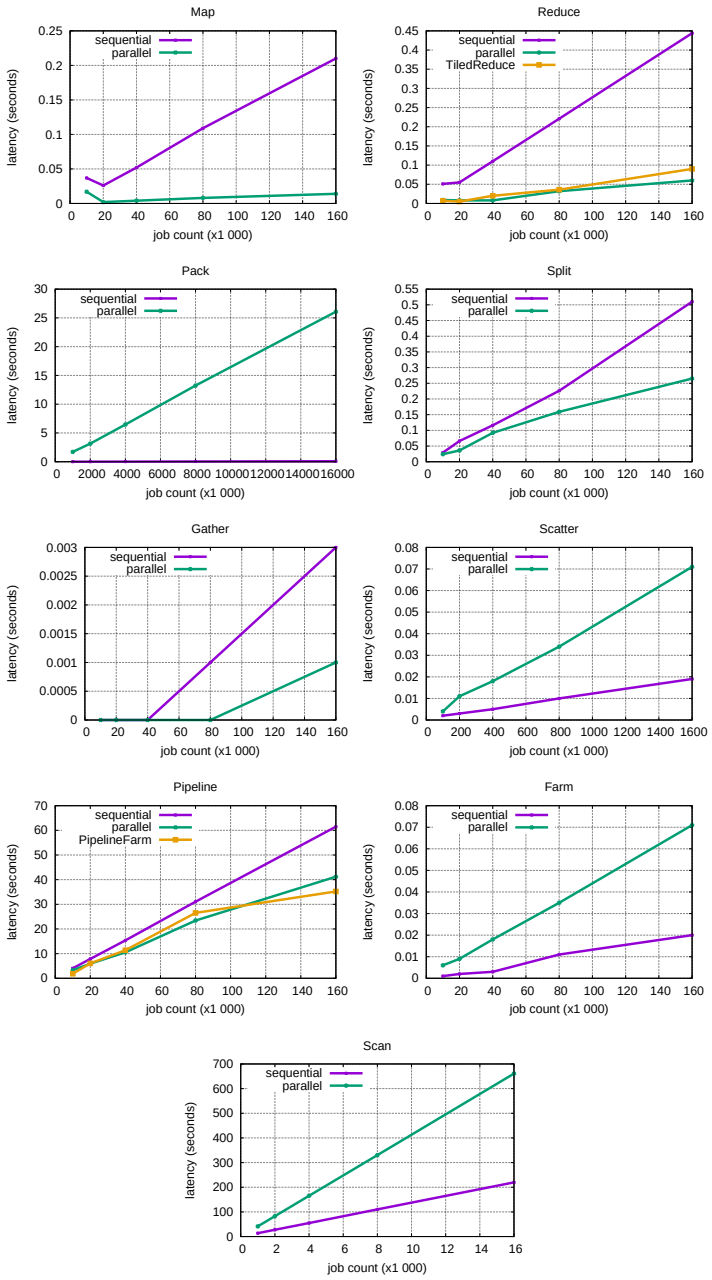

Figure 1. Latency of the algorithms.

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

4.2.1. Scatter. The parallel version of the Scatter pattern shows a noticeable worse computational performance when compared to its sequential counterpart as seen in Figure 1. This difference in performance seems counterintuitive, since parallel Scatter, being embarrassingly parallel (is trivially divided into parallel tasks), is implemented in the same way as the parallel Map pattern. Indeed, theoretically, the parallel Scatter should execute in less time than sequential Scatter. The cause for this result lies in the hardware, more specifically in the hardware memory caches. In a multi-processor system, each processor has its own individual cache (Level 1 cache), and they all share a common, slower cache (Last Level Cache). Everytime a processor reads a value from memory, a block of memory, representig the entire neighborhood of that value, is brought into the common cache, and the into the processor's individual cache. If one processor writes into a value stored in its individual cache, the entire block to which the value belongs, is marked as *dirty*, meaning it has been altered. If any other processor contains the dirty block in its individual cache (even if it does not intend to access the dirty value specifically), that block is invalidated, and must be reloaded from memory. The act of reloading a block from memory is very slow when compared to a cpu's clock cycle, and induces slowdown on the execution of a program. The more processors there are, accessing and writing to the same memory blocks, the more noticeable the effects of cache invalidation across the execution of a program. We used the tool *perf* to extract data about cache accesses, for both the parallel and sequential Scatter implementations. We observed a large increase in the number of accesses to the last level cache for the parallel implementation when compared to the sequential implementation. Indeed, an execution of parallel Scatter accessed, on average 40 times more to the last level cache than an execution of the sequential version for the same input. This is evidence of cache invalidation, which is the main factor behind the results observed.

5. Conclusion

The conclusion goes here.

Acknowledgments

The authors would like to thank...

Comments

References

[1] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

[2] 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.

[3] 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.

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