Studying Speedups: Parallel Patterns for Performant Programs

Pedro Camponês, Rafael Gameiro Departamento de Informática
Universidade Nova de Lisboa
{p.campones, rr.gameiro}@campus.fct.unl.pt

Abstract—Throughout the years, computers' capabilities have been increasing. However, computers can no longer grow at the same speed as physical limitations on components size or the generated heat is too higher to me cooled using the conventional methods. A solution to improve the hardware performance of computers was to increase the number of CPU cores rather than the power of each core. By carefully separating the execution between the executions threads, a process could achieve its maximum performance. In this report we present several parallel patterns used to maximize a program performance using the CPU cores. These patterns were implemented and their performance was measured by comparing the parallel implementation with a sequential one. Different scenarios were taken into account in order to conclude how these patterns react in those conditions.

1 Introduction

Ever since their inception, computers' capabilities have been ever increasing. The supercomputer CRAY-1, released in 1976, was a large machine specialized in the computation of mathematical operations. It could compute over 80 million floating point operations per second [1]. The enormous computing power of CRAY-1 pales in comparison to the modern smartphone, with the most recent models outperforming CRAY-1 by a factor of over a thousand.

The increasing computational power of computational devices can be attributed to Moore's law, which states that the number of transistors in a microchip double every two years. An increasing number of transistors leads to the ability to perform logical operations on a faster rate and to have more memory. The rate at which transistors are increasing in microchips has been dwindling as its increase is presenting diminishing returns. Today, a transistor is about 70 silicon atoms wide, and further reductions in transistor size won't be possible as quantum tunneling problems surge which aren't solved by conventional computer architectures [4]. One other reason why computers have been increasingly faster throughout the years is the maximization of clock speeds, allowing a CPU to perform more operations per second. There is however a problem with the reliance on clock speed; as the clock speed increases, more heat is released by the CPU. At a certain threshold, its not viable to increase clock speeds as cooling a CPU through conventional methods becomes impossible [3].

A solution found to increase the hardware performance of computers despite the presented restrictions is the increase of CPU cores rather than the power of each core. With this solution, computer programs can benefit increased performance by having different parts of the code being processed by different cores. As such, the performance of a program is highly intertwined with how prone its code is to be separated among threads. A parallel design pattern is a parallel solution to a common operation in programs. Several parts of a program can be converted to adhere to these parallel patterns, which results in their importance in assuring a fast execution of parallel programs [3].

In this report several parallel patterns were implemented and the increase in performance using a parallel implementation rather than a sequential was measured. These measurements take into consideration, the input size received by the pattern, memory access by each thread, the overhead in the creation of software threads, among other considerations.

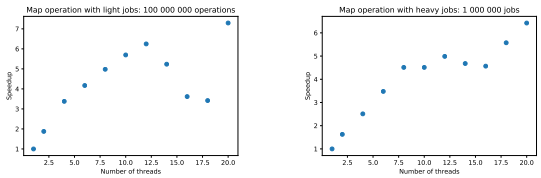
The patterns were implemented in the C language with the aid of Intel's OpenMP library for paralelization. The performance tests were run on a dedicated server with four *AMD Opteron 6272* CPUs, each with 16/16 cores; 64GB of memory; and 2 x 1 Gbps network bandwidth.

The distribution of the workload in this assignment is presented in section 2.

Section 3 is concerned with the performance of the Map pattern and how it can be used to attest Amdahl and Gustafson-Barsis' laws by being embarrassingly parallel. The Stencil pattern is also presented. Stencil is a sub-type of map. In this pattern, several functions are applied to produce an output, as such it is used to experiment with the code-fusion and cache-fusion optimizations for the Map pattern.

Section 4 concerns the patterns that aggregate input values, perform an associative operation over these values, and produce outputs resulting from these operations. A particular emphasis is placed on the Scan pattern as it is a pattern whose sequential and parallel implementations differs immensely.

Data reorganization patterns are presented in section 5. These patterns change process an input array through a filter to produce an output array where the original input



(a) Light Operations (b) Heavy Operations

Fig. 1: Speedup variations in map pattern

is reordered, some elements may be repeated and others removed. Particular focus has been given to the Scatter pattern as its performance is very dependent on specific characteristics of the values given as input and the filter; thus having implementations which are optimal for some circumstances but not for others.

Section 6 is related to the Pipeline pattern. This pattern follows a consumer-producer relationship where conceptually all stages are active at once, and data flows through the different stages until it reaches the last stage.

The final Section is concerning the Farm pattern. This patterns, typically known as master-slave pattern, creates and launches auxiliary workers who will perform operations in order to facilitate the execution. As soon as a worker has finished, its value can be outputted.

2 Work Distribution

An equitable division of workload was applied to the two authors of the report. Each author implemented four of the eight mandatory patterns. Pedro implemented: Map, Reduce, Gather, and Scatter; whilst Rafael implemented: Scan, Pack, Pipeline, and Farm. The additional Stencil pattern was implemented by Pedro, and the scripts used to run the tests in the cluster, as well as the handling of the cluster itself has been performed by Rafael. On the writing of the report, each element wrote the content relating to the patterns the mentioned element implemented. The introduction was written by Pedro and the conclusion by Rafael. Both elements adhered and enforced the correct use of git workflow. To conclude this section, the authors propose that in the evaluation process it should be considered that each element has contributed 50% of the resulting work.

3 Functions: Map and Stencil

The Map pattern receives a series of inputs and for each performs a bound set of functions, the result of which is stored in the output. The input elements and functions are independent amongst each other, as such the map pattern is embarrassing parallel, meaning that the work can be evenly split up between threads without concern for synchronization and there are nearly no cache conflicts. For these reasons, the map pattern presents the highest possible speedups of the tested patterns. According with Ahmdal's law [2], the sequential part of a program limits the maximum speedup that can be achieved. It's implied that the smallest the sequential part of a program is, the greater the maximum speedups achievable. Fig. 1 presents the speedups achieved with the map varying the number of threads running the program. The test in which the operations of the map were computationally lighter presents a smaller speedup than the test in which the operations were heavy. In the heavy operations test, the amount of parallel work is superior. Interestingly, there is a spike in the performance of the map pattern when using 20 threads. This phenomena is designated super-scalar speedup and its related to the alignment of values in cache with the threads; such that the threads rarely if ever read or write values that aren't cached.

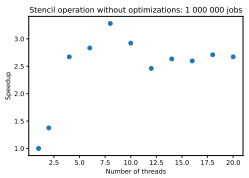
The Stencil pattern is a sub-type of map in which several input values are used to produce an output value. Unlike with the reduce pattern, the number of outputs will be equal to the number of inputs. As such, inputs are subject to repeated operations. By receiving several input values to generate an output, a reduce is applied upon these. This pattern is a relevant example on how the use of simple patterns can be combined and generalized to produce more complex solutions. Three implementations have been created for the Stencil pattern in order to attest optimizations to the Map pattern, when a series of functions must be applied to the same inputs. Given that Stencil is a sub-type Map, the improvements present in the Map pattern ought to be reflected on the Stencil. One of the implementations makes use of both codefusion and cachefusion; other makes use of only codefusion; and the remaining has no optimization.

Code fusion is achieved when a thread executes a series of functions on a set of inputs, and when the second layer of functions starts executing, the thread still has the first layer results in cache. Cache fusion is achieved when the sequence of functions to produce an output is executed completely before passing to the next input value.

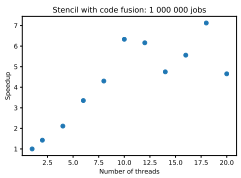
Fig. 2 presents the speedups attained in the Stencil pattern using the optimizations described. Unsurprisingly, the implementation without optimizations has the least amount of speedup increase as the thread number increases as well. The optimized versions present great speedups, with code and cache fusion achieving the highest speedup. It is however relevant to consider that the use of code fusion by itself places its implementation close to both optimizations' implementation, demonstrating the efficiency code fusion induces in performance of a pattern.

4 Collectives: Reduce and Scan

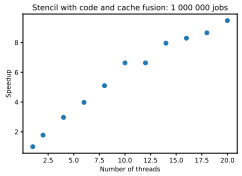
Collectives are a type of parallel patterns that aggregate several input values to produce an output. The resulting output will depend on the operations done over the input elements. In order to benefit from a parallel implementation, the operations performed must be associative, else wise, only a serial implementation can respect the order of the operations. The reduce pattern produces a single output from the inputs received. A particular type of parallel reduce has been implemented; it is designated *Tiling*. The implementation requires the division of the input into several adjacent segments of equal size, hence forward designated tiles; the number of which is equal to the number of threads executing. Each thread will then apply a reduce operation in its corresponding tile and store the contents on a auxiliary output. A final sequential step is performing a reduce operation over the threads' auxiliary outputs. This implementa-



(a) No optimization



(b) Code fusion



(c) Code and cache fusions

Fig. 2: Speedup variations in stencil pattern

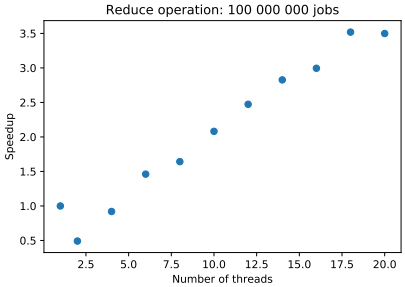
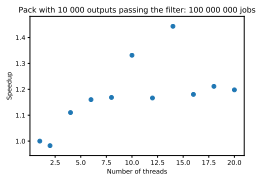


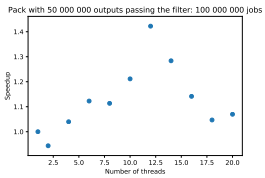
Fig. 3: Speedup variations in reduce pattern

tion is the fastest possible parallel implementation possible of the reduce pattern. However, in some operations, if using floating point inputs, it's possible that data will be lost on account of differences in precision. This problem was also present and exacerbated in the sequential version. A solution for this would be to create more fine grained reduces within each tile; that is to perform tiling within a tile, but using a single thread to handle each sub-tile sequentially. As may be inferred by the analysis of the algorithm, the reduce operation is not embarrassingly parallel, and as such, the speedups achieved don't follow the bounds set by Amdahl and Gustafson-Barsis as closely as the map pattern. Fig. 3 demonstrates the speedup of the reduce operation as the number of threads increases. The speedups attained in the Reduce pattern pale in comparison with Map or Stencil, an expected result which is corroborated by the data measured.

The scan pattern computes all partial reduction of an input sequence. Which means, for every output position a reduction of input until that point is computed and stored [3]. The implemented parallel algorithm is named *three-phase scan*. As the name states, the algorithm execution consists in three phases. The first one, the input is split among the threads and each one performs a reduction over the respective tile. The second phase will be a scan where the input values are the ones resulted from the first phase. Lastly, a scan operation is performed in each tile. For each tile except the first one, the initial value is the result of the scan performed in phase 2. As previously stated by the



(a) Output size 10000



(b) Output size 50000000

Fig. 4: Speedup variations in pack pattern

reduce pattern, in operations using floating point inputs, its possible that some data will be lost because of precision differences. The same solution model could be applied to this algorithm, by further splitting each tiling assigned to each thread. This way, a more fine grained phase would be applied to segments to the tiling and then a sub-thread would re execute the phase but this time using the results obtained from the first thread as input. Scan, just like reduce pattern, is not an embarrassingly parallel pattern and therefore it does not strictly follow the Amdahl and Gustafson-Barsis Laws like the map pattern. The experiments conducted in this pattern were the same made in the map pattern, testing its performance with a light operation and then with an heavy one.. With this we can conclude that the type and heaviness of the operation the pattern will apply to each input element will directly influence the execution times. As simple operations cannot bring great difference in execution times, while an heavy one can greatly impact the algorithm performance.

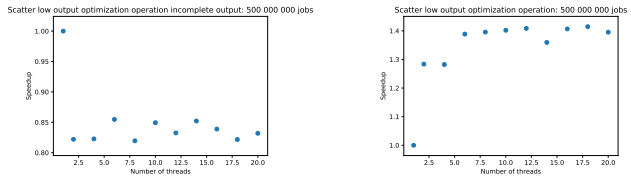
5 Data Reorganization: Pack, Gather, Scatter

This section concerns patterns that reorganize data so it can be processed by other parts of the program and possibly other patterns. Three data reorganization patterns will be explored: Pack, Gather, and Scatter.

The Pack pattern applies a filter to an input array. The elements of array that pass the filter are placed in an output array in order. This pattern was implemented by dividing the input in several adjacent segments and performing a sequential Pack to each segment, placing the outputs in auxiliary output arrays, whose values will then be copied sequentially into a final output array. Fig 5 shows the speedup achieved by the Pack pattern as the number of threads increases. Two test were performed, one in which there were few inputs passing the filter, and other where half the input passes the filter. The speedups are nearly identical in both tests; this indicates speedup gained from the second parallel part of the implementation (the copy from the auxiliary outputs) is identical to the speedup achieved by processing the filter in parallel.

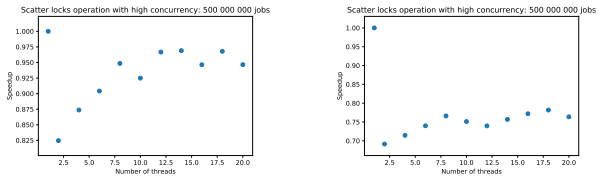
The Gather pattern reorders the elements of an array, and may remove some. Associated to the output array is a filter array with the same length. Each element of the filter array contains the input index of the element to be placed in the output.

The Scatter pattern is the reverse of the Gather pattern, such that the filter array is associated with the input rather than the output. Every value in the filter indicates



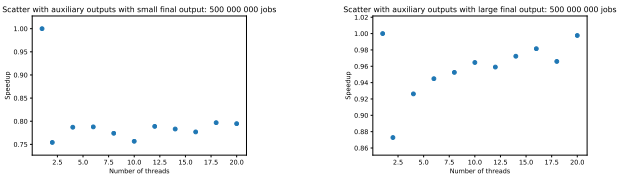
(a) Worst case (b) Best case

Fig. 5: Speedup variations in scatter pattern optimized for low output



(a) Low concurrency (b) High concurrency

Fig. 6: Speedup variations in scatter pattern with locks



(a) Low output (b) High output

Fig. 7: Speedup variations in scatter pattern with auxiliary outputs

the output index where the corresponding input is to be inserted. The parallel execution of this pattern may lead to race conditions, as two inputs may be written on the same output position. Several implementations of the scatter pattern were made, each being optimized for specific configurations of the output. All implementations to be presented are deterministic in their output. So, for a given input, the output will always be the same, independently of the threads' scheduling.

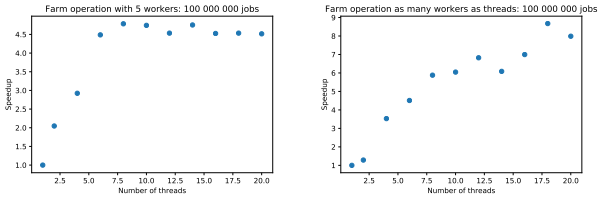
The first implementation of the scatter pattern consists on using the Reduce pattern to discover the size of the output array, and then performing a sequential Scatter over the filter, stopping midway through the operation if all the elements in the output have been filled. The Reduce is required because knowing the length of the output is needed to be able to detect when all elements have been filled. This implementation is exceptional for small outputs, as it is able to leave the filter early without incurring in the synchronization overheads a parallel solution has. Fig. 5 demonstrates the speedups achieved using this implementation for small outputs which become completely filled and for outputs that have at least one unassigned entry. It can be seen that when the output is completely filled the performance of this implementation far outpaces the base. The opposite occurs when the output is incomplete. The speedup is not drastic in either situations, because the bottleneck of the operation is the sequential creation of the filter.

The second implementation devised uses locks to avoid race conditions. A Reduce in executed to determine the length of the output, with this, an array can be created to indicate which thread has written in which output (or -1 if no thread has written on it). The filter is segmented into parts, and each thread will execute a Scatter into the output array. The Scatters performed by the threads differ from the sequential version because a thread will only write to outputs which were written by a thread of lower index; and to write in the output, the thread must first acquire a lock. The size of the output is a crucial factor in the performance of this implementation. Fig. 6 presents the speedup achieved using a parallel implementation of the lock solution based on the number of threads. This solution is not performant for neither small and large output arrays, but especially for small arrays. These results appear contradicting, as less write operations would occur in the small output array, given that a value would only be written if a smaller thread index had written it before. In the large output array, more write operations would be performed. Although it's unlikely the threads would have to wait to acquire a lock, acquiring it results in the execution of several instructions. As such it was expected that the large output array would have lower speedups. The computational hindrance presented by the acquisition of the locks may be the main factor which degrades this implementation's performance.

The final implementation created does not use locks and is optimized for large outputs. First a Reduce is made to determine the length of the output. Then the filter is segmented and each thread will perform a Scatter from one of the segments into an auxiliary output array. After the scatters are complete, the output array will be traversed in parallel, and for each index of the output array, an element from the auxiliary outputs will be retrieved and written. Fig. 7 shows the increase in speedups using both small and large output arrays. The large output test has greater increases in speedups because the final parallel operation in the small output array test is too short to be relevant. This final implementation has a drawback compared with the remaining which is the space required to store the auxiliary outputs. Each auxiliary output will have the same length as the final output, as such, this implementation is hindered by using several threads and by having a large output array, which is what this implementation was optimized for.

6 Specialized workers: Pipeline

The pipeline pattern consists in a linear sequence of stages, each performing an operation to its input. The data flows from the first stage until the last one, and the input of one stage is the output of the previous one [3]. There are two basic approaches to implement a pipeline, and those two were done to better analyze the performance in a parallel environment. The first implementation, named *Bind to Stage*, bounds a worker to a stage and processes each item as they arrive. The second one, named *Bind to Item*, consists in assigning a worker to an item. That way, contrary to the first implementation, the worker moves through stages while carrying the item.



(a) 5 workers (b) same number of workers as threads

Fig. 8: Speedup variations in farm pattern

To evaluate the performance of each implementation, a set of tests were made to compare the speedup of the pattern when the number of stages vary. Even though in the code submission there weren't tests made to the second implementation, we saw the necessity to do it in this report so the designed tests were done to both implementations. In the first implementation, each worker is limited by the dependencies between workers and because of that, a worker can only execute its operations to an item as soon as the previous workers had executed the assigned operation to the same item. In the second implementation, the workers aren't bounded between them since each worker executes the entire pipeline with each item it receives as input. Despite that, both implementations seem to display the same performance even when the number of threads is increased, performance that it is lower than the sequential version of the pattern. However, we can conclude that as the number of stages increases the time needed to conclude the execution also increases, since the workload is bigger.

7 Generalized workers: Farm

The farm pattern, typically known as master-slave pattern, assigns a task consisting in an operation or sequence of operations to a group of workers, so that each worker executes the same operations and only return its result to the master. The implementation created assigns a single master the responsibility to generate the tasks, so that the remaining threads would be the workers and execute the generated tasks. A test was made to compare the performance of the algorithm when a different number of workers were assigned. Fig. 8 displays the test results, and as we can see, the more the workers assigned the faster the program is executed. Because this pattern strongly resembles the map pattern, where in this case each worker would execute an operation to each input item and then output it, we can assume it is embarrassingly parallel and therefore, strongly follows the Amdahl and Gustafson-Barsis Laws where until it reaches a given threshold the more the workers the better is performance will be.

8 Conclusion

In this report we have presented a comprehensive analysis of parallel patterns. We have enlisted several patterns used nowadays to maximize the use of threads to improve performance. In our work, we analyzed well-known implementations of some patterns, and in others we presented our own solution. We presented a brief description about each studied pattern and then a performance analysis related to our implementations. In some patterns, we compared its theoretical concept and implementation with the Map pattern which is a embarrassingly parallel pattern, and therefore strictly follows the Amdahl law and Gustafson-Barsis law.

Acknowledgments

The authors would like to thank the colleagues Pedro Valente nº 50759 and Rafael Sequeira nº 50002 their contribution during the implementation of scatter, by suggesting the use of OpenMP version of reduce pattern to optimize the performance. Also, the authors thank colleagues Pedro Valente nº 50355 and Diogo Tavares nº 50309 for their suggestion to use a reduce over the filter in the scatter pattern, to compute the output size.

References

[1] Cray Research, I. The cray-1 computer system. https://archive.computerhistory.org/resources/text/Cray/Cray.Cray1.177.102638650.pdf, 1977. Accessed 17/05/2020.
[2] Gustafson, J. L. *Amdahl's Law*. Springer US, Boston, MA, 2011, pp. 53–60.
[3] Mccool, M., Reinders, J., and Robison, A. *Structured Parallel Programming*. 01 2012.
[4] Seabaugh, A. The tunneling transistor - ieee spectrum.