CP 1920 Project Report

Tomás Pessanha, *N 41774,*

Abstract—This is the project report for the assignment of the course concurrency and parallelism, the assignment was to implement a set of parallel programming patterns using OpenMP [1] and evaluate them.

Index Terms—computer engineering, concurrency, parallelism, parallel patterns, OpenMP.

1 Introduction

TODAY'S processors have several cores and many times support for even more threads. Given this fact it is in our best interest to use them to improve our program performance and scalability. This is a decision the industry is more and more choosing to take, simply making a single core faster is much harder than having more cores in the same processor.

The objective of this assignment was to implement a parallel version of the following patterns (Map, Reduce, Scan, Pack, Gather, Scatter, Pipeline, Farm) and optionally others and evaluate them using OpenMP [1].

1.1 Methodology

The computer used for testing has six cores, and twelve threads with Simultaneous MultiThreading (SMT [2]).

Every table shows time in seconds unless specified otherwise.

May 18, 2020

2 Patterns

2.1 Map

The map pattern is a simple function applied to each element of an array.

2.1.1 Implementation

To parallelize this pattern the most intuitive approach was used, the array was divided by the amount of threads and each threads worked on it's section of the array.

2.1.2 Evaluation

The map pattern is a very good pattern to parallelize, in theory it should scale linearly but in practice it never does, still it scales very well since the management cost is almost nonexistent, the only extra work the program has to do is the creation of the threads themselves and the division of the array to map.

In table (1), we can see the time it took for each run of the map function of adding one to each value. We can see that this patterns scales very well with physical cores, but a much smaller change if more software threads are used than physical cores available. The job in question is simple enough that having the extra threads, even with simultaneous multithreading of the processor active we see little but still measurable benefit. Adding even more threads made the program run slower.

TABLE 1
Map evaluation

	Input:			
cores/threads	**10**	10^3	10^6	10^8
1/1	0.00000	0.00001	0.00299	0.43981 (1x)
2/2	0.00007	0.00007	0.00198	0.22524 (1,95x)
4/4	0.00009	0.00011	0.00146	0.13100 (3,36x)
6/6	0.00014	0.00014	0.00130	0.11024 (3,99x)
6/12	0.00013	0.00016	0.00127	0.09962 (4,41x)

Now the question of "what if we make it harder?". Instead of using the add one worker, lets use a slower worker and see the results in table (2) in this one we locked the input at 10^6.

TABLE 2
Map hard work

cores/threads:	1/1	2/2	6/6	6/12
time (in seconds)	1.79377	0.91899	0.33951	0.15770
Speedup	1x	1.95x	5.28x	11.37x

Here the story changes, the performance improvement is almost linear, since relatively more work is being done than taking care of thread management and function calling, still using more than the given twelve threads sees no improvement.

2.2 Reduce

The reduce pattern consists of transforming all the elements of an array into a single one, 'reducing' the array into a single value.

2.2.1 Implementation

The operation used has to be associative in order for the parallel implementation to work.

First an array of size equal to the number of threads it's created, then the original array is divided by the threads and each threads does a serial reduction on his part of the array, adding the result to the previously mentioned array. A final

• *T. Pessanha is studying Computer Engineering at the NOVA University of Lisbon, N, 41774.*
E-mail: t.pessanha@campus.fct.unl.pt

Delivered May 18, 2020

serial reduction is done on the array after all threads are finished, since the operation is associative the order does not matter, this implementation takes advantage of this to parallelize the pattern.

2.2.2 Evaluation

The reduce pattern should also scale very nicely as there is once again little management needed. And in table (3) can be seen just that, the scalability of this program is indeed very good with physical cores, almost linear, we slow down once we use more threads than physical cores, but we still get a considerable speed up using the SMT of the processor, but once again we stop getting any benefit after twelve threads.

TABLE 3
Reduce evaluation

	Input:			
cores/threads	**10**	10^3	10^6	10^8
1/1	0.00000	0.00001	0.00285	0.28056 (1x)
2/2	0.00002	0.00002	0.00224	0.14063 (2x)
4/4	0.00004	0.00004	0.00164	0.07471 (3,75x)
6/6	0.00005	0.00005	0.00107	0.04974 (5,64x)
6/12	0.00011	0.00110	0.00209	0.03754 (7,47x)

Now for the hard task, in table (4) ca be seen the results of locking the input at 10^6 and running an harder job than simple sum.

TABLE 4
Reduce hard work

cores/threads:	1/1	2/2	6/6	6/12
time (in seconds)	2.65576	1.34958	0.46740	0.24169
Speedup	1x	1.96x	5.68x	10.98x

The speed up is almost the same, with one exception, the speedup when using all twelve threads is much better now. This seems to be a tendency, when the work is hard the extra threads help in a very considerable way, when the work is small the extra threads just get in the way, this is a very interesting fact, let's see if the other patterns have the same effect.

2.3 Scan

The scan or prefix-sum pattern as it is also called, is the cumulative sum of the input at each given step, like shown in (1).

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 + x_1 \\ &\dots \\ y_n &= x_0 + \dots + x_n \end{aligned} \tag{1}$$

This patterns has an obvious problem if you want to parallelize it, it looks like there is a dependence between the current step and the previous one, as you can only get the value of step i if you have step $i-1$, we can get pass this problem in a non intuitive way using a binary tree with intermediate values.

2.3.1 Implementation

First the tree needs to be created, so we need to reserve an array that is going to be our tree of size $(n*2)-1$, with n being the amount of inputs, but why this size?

This is the perfect size to have two proprieties, the first is that the leaf nodes of the tree are the same as the inputs, the second is that every node has either two or zero children this makes the job easier.

The implementation of the parallel version of scan consists of the following steps:

1) Allocate the space for the tree.
2) Copy the input to the leafs of the tree.
3) The up pass, go through the tree bottom to top, so that each node has the value of the sum of it's children, and the leaf nodes have the input.
4) The down pass, go through the tree top to bottom, building the intermediate values along the way.
5) Copy the leafs to the output array.

2.3.2 Evaluation

Scan is the first pattern with a considerable management time until now. Since the program needs to build a tree structure the space needed is $(4*n-2)$ just for the tree, and each node is not free.

Table (5) shows the evaluation results for scan.

TABLE 5
Scan evaluation

	Input:		
cores/threads	10^3	10^6	10^8
1/1	0.00001	0.00321	0.44617 (1x)
2/2	0.00007	0.01240	1.19204 (0.37x)
4/4	0.00008	0.00786	0.68997 (0,65x)
6/6	0.00009	0.00753	0.56413 (0,79x)
6/12	0.00132	0.00731	0.55037 (0,81x)

There seems to be an issue here, we are getting worse performance than the serial version. Do we need more cores?

The performance does seem to be improving so at least it scales but even if that is the case the scaling is less than optimal to say the least. There seems to be way too much overhead on the management part of this program, if so most programs should probably run the sequential version unless they have a lot of cores and a lot of data to run.

What about if we make it harder again? Table (6) has once again the input locked at 10^6, and the the work is a much harder one.

TABLE 6
Scan hard work

cores/threads:	1/1	2/2	6/6	6/12
time (in seconds)	1.69916	3.04981	1.25246	0.56500
Speedup	1x	0,56x	1,36x	3x

Well it's not the amazing scalability one might have wanted but it is there. And yet again it seems like multi-threading helps a lot, wait a minute even more than real threads? It seems so, the tests were redone and the result is the same. But why? The cores are memory hungry, and spend some time waiting for the memory they need, this

issue is alleviated with SMT. When one core is working on something, the other part of the core is getting things ready for the next one so it is a much more efficient way of working, this is why SMT was introduced and seems to be working amazingly well in this case.

More than twelve threads slows down the program.

2.4 Pack

The pack pattern functions like a filter, given an array $input$ and a Boolean function $f(e)$, produce an array $output$ containing only elements e such that $f(e) == true$.

This pattern is hard to parallelize because we only know where each element goes into the array if we know the previous elements. One solution is to use the previous pattern (scan), this will be explained in the next implementation section.

2.4.1 Implementation

Normally we would have to run a map using the filter function in order to get a bit-vector for the elements that are going to be included and the ones that are going to be removed. But here a bit-vector is given as input so this step is ignored.

Next a scan is performed on this bit-vector called $filter$, we will call the output of the scan $bitsum$.

And now the size of the output array is at the last position of the $bitsum$ array.

Finally we go through the $filter$ array, for every i such that $filter[i] == 1$, we copy the value of the original array on position i and place it on the $output$ array at the position $bitsum[i] - 1$.

An example of this can be seen in (2), with a filter that returns on values above five.

$$\begin{aligned} input &: [7][4][2][8][1][6] \\ bit-vector &: [1][0][0][1][0][1] \\ bitsum &: [1][1][1][2][2][3] \end{aligned} \tag{2}$$

2.4.2 Evaluation

The pack pattern needs to run a scan and scan was already slow, this will make it very slow. Table (7) shows the basic evaluation results.

TABLE 7
Pack evaluation

	Input:		
cores/threads	10^3	10^6	10^8
1/1	0.00001	0.00472	0.51502 (1x)
2/2	0.00003	0.00887	0.78464 (0,66x)
4/4	0.00004	0.00794	0.74358 (0,69x)
6/6	0.00004	0.00806	0.74923 (0,69x)
6/12	0.00096	0.00873	0.70556 (0,73x)

Well that doesn't seem like an improvement at all. Not only is the parallel version getting worse performance than the sequential version, but the scaling is non existent.

But exactly the same? Why ? Something must be wrong here, if we look at the code we see a little function called $scan_seq$, this is a sequential scan, the reason a sequential scan is here and not a parallel one is that the work to be one is a simple sum of numbers, so it most likely will run faster in sequential mode, but this makes this part of the program a bottleneck. It will never run faster than the time it takes to run a sequential scan, and that is a lot of time.

So is this pattern useless? In this implementation and with this amount of cores yes, the solution for this is to remove the bottleneck the scan creates. One of the ways is using a parallel scan, but in order to catch up to the sequential scan it need a considerable amount of data and cores to do so, maybe with a better implementation is would be possible, as it stands it with the amount of cores available it will never be faster than the sequential pack.

2.5 Gather

Gather pattern receives a collection of indices called $filter$, then reads the value v of that index in the $input$ array, and copies that value to the corresponding position such that for each i, $v = filter[i]$ and $output[i] = input[v]$.

2.5.1 Implementation

The parallel implementation of gather is very straightforward, the $filter$ array is divided between the threads, each thread just have to then copy the correct value as explained the the previous section, to the output.

2.5.2 Evaluation

The gather pattern is very similar to the map pattern, in that has almost no management and should scale similar to a map.

Table (8) shows the evaluation of gather pattern.

TABLE 8
Gather evaluation

	Filter and Input:		
cores/threads	10^3	10^6	**filter (10^6) input (10^8)**
1/1	0.00003	0.01735	0.02715
2/2	0.00004	0.01502	0.01552 (1,75x)
4/4	0.00005	0.01326	0.01337 (2,03x)
6/6	0.00006	0.01295	0.01096 (2,47x)
6/12	0.00039	0.01305	0.00994 (2,73x)

In the gather pattern it seems some bottleneck exists again, we should see a more or less linear speedup, but we don't. After checking with perf what do we see, lots and lots of idle time. This is where this pattern differs from map, in map the memory access were many times less and more sequential, here the memory access is random, so it is slower causes much more conflicts and can overwhelm the memory.

2.6 Scatter

The scatter pattern is the inverse of the gather pattern, the pattern receives the $input$ array and the $filter$ array, what the pattern does is for each index i of the array $filter$, that from the value of the same index i form the $input$ array, and place it in the $output$ array at the position $filter[i]$. So to summarise it does $pos = filter[i]$ then $output[pos] = input[i]$.

2.6.1 Implementation

Scatter pattern has a very clear race condition that needed to be taken cared of. There is a possibility that several threads want to write to the same memory space, and even worse if the write instruction is not atomic the final value can be corrupted and simply wrong.

The chosen implementation is one that doesn't say which of the values stays in the memory position. The implementation assures that the first thread that reaches the memory gets to write the final value for that position. This is achieved with the use of locks, a lock is created for every memory position of the filter, once a thread locks a specific lock, it never unlocks it, so the value of that thread is the final value.

If a thread finds a locked position it just ignores it and continues.

2.6.2 Evaluation

The implemented version of scatter uses locks, one for each input, this is to avoid race conditions. But this also adds management cost, luckily is not much, lets see table () that contains the results of the evaluation and analyse the results.

TABLE 9
Scatter evaluation

	Filter and Input:		
cores/threads	10^3	10^6	10^8
1/1	0.00003	0.01608	2.51277 (1x)
2/2	0.00006	0.03122	3.58888 (0,70x)
4/4	0.00006	0.02305	2.74933 (0,91x)
6/6	0.00007	0.02039	2.45416 (1,02x)
6/12	0.00034	0.02026	2.39661 (1,04x)

While some speed up can be noticed the scalability is not great, this is the same problem that gather suffers, plus the fact that it needs to check locks every time. Can't say this implementation is amazing but at least it shows some scalability even if little.

2.7 Pipeline

The pipeline patterns works much like a real pipeline in factories. There exists n threads each working on a specific part of the work given only to that thread, no two threads have the same work. Much like an incremental process the data is passed from thread to thread until all threads worked on it and it is completed.

2.7.1 Implementation

The implementation of this pattern uses an array of pointers to integers where each of this pointers saves the amount of work done by the thread, the first position is the total jobs to be done. Example $[10][5][1]$ signifies that there is ten jobs to do, the first thread did five and the second thread did one. Then each thread enters a loop that checks if his own jobs done is less than the total jobs to do, if not then it checks if his own jobs done is equal to the amount of jobs done by the previous worker, and if not that means he has work to do.

While this implementation would work with a normal array of integers instead of an array of pointers to integers, after some testing a speed up was noticed in more intense scenarios while using the pointers, the reason is that the threads are trying to access and change the same cache lines and slowing each other down, this is called false sharing. The problem with this improvement is that the management cost is superior, but since scalability is the objective here, this is the superior option.

2.8 Evaluation

The pipeline pattern is an interesting one to analyse since it has a predefined number of threads to run. Table (10) shows the results of the pipeline evaluation with six workers.

TABLE 10
Pipeline evaluation

	Input		
Version	10^3	10^6	10^8
Sequential	0.00002	0.02497	2.47259
Parallel	0.00281	0.05427	6.80970
Sequential_2_Hard	0.00558	5.60970	2
Parallel_2_Hard	0.00594	3.39969	2

If the work is easy, sequential is faster than parallel, but when out of the six works, two of them are doing a hard task we can clearly see improvements in performance. And the more workers working on hard tasks the better the improvement. Pipeline is a pattern that does well if all the workers have a similar amount of work and is hard work.

2.9 Farm

The farm patterns also known as master/slave pattern, works by having a master thread split the jobs to be done between a fixed amount of slaves (not threads), you can have more slaves than threads and you can have less slaves than threads, the amount of slaves is arbitrary, then the actual threads will take on the work of zero, or more slaves.

2.9.1 Implementation

The OpenMP [1] task directive is used here to achieve this pattern. The master thread splits the work by launching OpenMP [1] tasks of the divided work to be worked by the slave threads, the interface of the farm function receives $nWorkers$, this is the amount of slaves. After the split is done the master thread joins the slaves to maximise performance.

2.9.2 Evaluation

The farm pattern uses a fixed amount of workers, and each thread takes the role of one of the available workers. Given this it makes sense to always use the maximum threads physically available to test it, in this case it's twelve.

So with twelve threads available and a fixed input size of 10000000 the table (11) shows the results while varying the amount of workers and not threads.

The cost of using tasks is clear, since the same work using a simple parallel map is slightly faster, but when given no choice this pattern still scales reasonably well.

TABLE 11
Farm evaluation

	Workers				
Version	**1**	**2**	**4**	**6**	**12**
Sequential	0.04364	0.04391	0.04409	0.04372	0.04439
Parallel	0.04180 (1,04x)	0.02755 (1,59x)	0.01893 (2,33x)	0.01420 (3,08x)	0.01218 (3,64x)

3 Conclusion

While the implementations of this patterns are not perfect, they served to show the advantages and hardships of parallel computing. It would be needed a lot more space and time to fully discuss them but this work should provide a basic understanding of how the patterns work, their strong points and their weaknesses.

4 Work Division

This assignment was done by a single element.

5 Comments

A mini lecture on how to profile parallel code with perf or any other tool would be appreciated if time allows.

OpenMP is easy to learn and use I don't know about cilk but I felt no hardships using OpenMP.

Youtube lessons help a lot, (specially when watched at 2x times speed turns 1h lessons into 30min) so might be a good idea to continue with those, ask the other students if they agree with me on this one I don't know how they learn the best.

Acknowledgments

The author would like to thank:

1) Pedro Pais (48247), for helping with scan question in piazza.
2) Filipe de Luna (48425), for helping with scan question in piazza.

References

[1] OpenMP Architecture Review Board, url: http://www.openmp.org
[2] Wikipedia article: Simultaneous multithreading , url: https://en.wikipedia.org/wiki/Simultaneous_multithreading