Parallel patterns using Cilk+

Implementation, performance and scalability

Hubert Olkiewicz (54888), Pedro Miguel and Sofia Afonso

**Abstract**— this work presents results of implementation of patterns for Parallel Programming using Cilk+. Eight patterns are considered: Map, Reduce, Scan, Pack, Gather, Scatter, Pipeline and Task Farm. Each algorithm implementation approach is described. Efficiency and as well as scalability is main concern of paper hence we have measured execution times of our algorithms implementations using 16 CPU machine, showing times depending on the number of used threads. Most challenging part was to not only parallelize algorithms but also make them as efficient as possible, and to make it so, sometimes avoiding complex data structures and moving towards simplest data types was yielding significant speedups. We have lerned that C compiler is that efficient that even previously C code gimmicks, which can be found at online website [1] were not reducing assembly code at all nor improving execution time. Also avoiding complex structures (in comparison to basic type array) showed significant speedups. Although not every pattern implementation showed efficiency improvements, for instance Task Farm due to function template, could not show any speed up increase since of minimal parallel task data granulation on contrary to overlay of worker dispatching functionality.

**Index Terms**— Parallel pattern. Parallel implementation. Parallel performance. Parallel scalability.

————————————————

1. Hubert Olkiewicz Author is with the Wrocław University of Science and Technology. E-mail: 200145@student.pwr.edu.pl
2. Pedro Miguel
3. Sofia Afonso

—————————— ◆ ——————————

# 1 Introduction

P

ARALLELISM nowadays is supported by all modern computers by for example: vector instructions, multithreaded cores, multicore processors, graphic engines and parallel co-processors. Even most of current mobile phones support parallelism. Although parallel programming requires vastly different approach than sequential programming to be effective.

We are showing our implementation approach to most common parallel patterns: Map, Reduce, Scan, Pack, Gather, Scatter, Pipeline and Task Farm, broadly describing each implementation of them, including performance tests. Cilk+ library is used to provide easier and higher level programming parallel implementation model which allows us automatically set up most efficient configuration of working threads, considering our CPU capabilities.

Most challenging part is to not only parallelize sequential algorithm but make it as fast as possible, which is not that easy considering usage of Cilk+ library. Cilk+ has strict requirements regarding usage of keywords. Specific rules have to be followed. For instance working with cilk\_for requires from developer to follow certain rules [2], to not only make cilk\_for work but be efficient as much as it can be.

First thing that we have learned is that compilers nowadays are really efficient. During map pattern experimentations we have decided to try to make adjustments in assembly code using firstly some C coding gimmicks [1] and then adjust assembly code by ourselves but neither attempts brought expected results. Secondly, parallelization task should be carefully adjusted to whole program. In some cases (ex. Task Farm) overlay of additional code is that big that parallelization has no real translation to better results. In that case parallelization task should take at least CPU heavy calculations to work over as parallelization supporting code overlay is. Thirdly, if you want to do efficient code, you have to operate on most basic structures. Based upon reduce, we decided to avoid usage of tree structure because you can effectively move around array just like it was a tree so after calculating proper cell number to access, we can get data in time on contrary to tree and time consumed on creating new nodes.

Results were a little dependent on imposed method template (it biased any parallelization speedup at Task Farm implementation, because of workers tasks data granularity). Nevertheless in map and reduce implementations, speedup using multiple threads was significant. Pack pattern showed minimal execution time improvement. Rest of algorithms were working equally or slower using multiple cores. It shows that parallelization itself was working (cause of execution time difference between one and multiple cores) but it yielded only CPU surplus calculations regarding, in most cases unused, threads.

# 2 Architecture / Model / Solution

## 2.1 Map

Reduce pattern description as well as rules are broadly described in [3].

**Parallel map algorithm:**

**Input:** Data array *src*, length of array *n*, function where stands for element in array

**Output:** Data array dest

What it does is basically traversing throughout whole *src* array while executing given function *f* on the element that he is currently pointing at, and saving its result in according position at destination array *dest****.***

## 2.2 Reduce

Reduce pattern description as well as rules are broadly described in [4].

Our implementation is nearly the same, as figure presented in [4 Fig. 3.9] shows, only saving order differs, where our algorithm gather data in first cell, and so all intermediate reduces are joining towards as well first cell.

Nature of this algorithm implies to use tree based structure, although using any structure overlays additional work for CPU (mostly creating new nodes). Our approach was to simulate tree structure, by traversing on given array of data in specific, tree like, way. This saves us some CPU cycles as well as intermediate dynamic memory. Although while using this algorithm it is given that every operation is made on source array, hence it is recommended to work on a copy of data if immutability of data is important.

**Reduce algorithm:**

**Input:** Data array *src*, length of array *n*, function where stands for element in array

**Output:** Element dest

Algorithm works as follows: – is number of intermediate cell reduces levels (for ex. Figure 2 shows 3 levels) required to reduce whole tree. Main loop (this loop can’t be parallelized since each level calculation needs to have previous level already reduced) is responsible for traversing throughout whole reduction tree. – is base array cell shift for each level required to access right cells for reduce function. Parallel loop condition is as follows: , and it controls how many iterations should loop make over distinct (the deeper level, the less iterations over array). Inside parallelized loop, cells to reduce are calculated, and over them is executed which returns reduction result. Ceiling function is used because this algorithm first loop begins with .

## 2.3 Scan

The Scan pattern, also known as the Prefix Sum Problem, illustrates a pattern that can be used in many problems, being some of them “minimum, maximum” is implemented using two steps.

NOTE: The Scan algorithm implemented in this project is only usable for input arrays of size 2^n. If any other value is used in the testing of the algorithm then (if not previously commented) will not allow the correct testing of the other algorithms, resulting in a Segmentation Fault.

This problem is known, just not resolved.

## 2.4 Pack

Very generally the pack pattern is used to eliminate unused space in a collection (what we will call filter).

The elements marked as “true” are kept while the rest is discarded. The elements that remain are placed in a contiguous sequence in the same orderl.

## 2.5 Gather

The gather algorithm receives as input a data collection and a filter array, and it filters it according to the indices of the filter, generating an output array with the same dimension as the filter. [3]

To parallelize this algorithm, cilk\_for was used because it allows performance to get better as the size of the input array increases.

## 2.6 Scatter

The scatter pattern receives a data collection and a set of indices as inputs, however, while gather reads the element, scatter pattern writes the input elements at the indices’ location. [3]

This means that we may have concurrent writes when we try to parallelize this pattern.

In order to parallelize the given code, it was used the cilk\_for keyword.

## 2.7 Pipeline

A pipeline pattern breaks down processes into tasks, and each task is responsible for completing one step of a calculation and then passing the data on to the next task and so on. [3] The tasks are workers and in each stage, a worker performs a calculation, updates the data, and passes it on to the next worker, and so on until the last stage.

## 2.8 Task Farm

Task Farm background is well described in [5].

This algorithm is really similar to map pattern with addition that there are workers. Each worker gets chunk of data to process from master, but since it is parallelized, worker can be accessed from multiple threads hence worker himself is critical section, so usage of collections like concurrent queue is highly recommended. Our approach uses ordinary list of numbers where implied that worker is currently , and – . Master is a loop which is giving out tasks with data to process. There was no need for collector since worker gets destination chunk of data.

**Task Farm algorithm:**

**Input:** Data array *src*, length of array *n*, function where stands for element in array, function accessing critical section of workers which returns worker (which at moment of enlistment should be set to ).

**Output:** Data array dest

# 3 Implementation

In all implementations there are several, repeating, code variables:

src – source array of elements

dest – destination array of elements

worker – function to be done over elements

nJob – size of given arrays

sizeJob – size of array type

Any additional auxiliary variables are described in algorithm implementation section.

## 3.1 Map

Basic implementation, just usage of cilk\_for is satisfactory to parallelize algorithm.

cilk\_for (register unsigned int i=0; i < nJob; i++) {

worker(dest + i \* sizeJob, src + i \* sizeJob);

}

## 3.2 Reduce

int treeLevels = ceil(log2(nJob));

for (long treeLevel = 0; treeLevel < treeLevels; treeLevel++){

long levelCellShift = pow(2, treeLevel);

long revertedLoopCondition = ceil(((nJob - levelCellShift)/levelCellShift));

cilk\_for (long i = 0; i <= revertedLoopCondition; i+=2){

long cell1 = i \* levelCellShift;

long cell2 = cell1 + levelCellShift;

worker(src + cell1 \* sizeof(sizeJob), src + cell1 \* sizeof(sizeJob), src + cell2 \* sizeof(sizeJob));

}

}

What is interesting is that providing inner loop local variables (cell1, cell2) vastly improved execution time. Also revertedLoopCondition is introduced because of cilk\_for loop rules which one indicates that loop counter shouldn’t take any other actions beside of incrementing (nor even be used in loop condition calculations).

## 3.3 Scan

In the first step to implement this algorithm we begin by building a tree bottom-up, and on the second step we traverse that same tree top-down.

For this we build a binary tree where the root has an attribute, sum, of the range [x, y] where x and y are the limits of the input, and we then fill up the rest of the tree by adding the “sum” values of each child to the parent. We proceed to do this until the tree is filled up. This is called the Up Pass (or upsweep). After this we can proceed to the second part. The Down Pass (or downsweep).

We now use the tree previously made to get the prefix sums using an easy fork-join computation.

Starting at the root, which is given the value of 0, we proceed down the tree each time passing to the left child the value already calculated and to the right child that same value plus the value “sum” of the parent node.

All of this will result in a tree where the leaf positions will correspond to the answer of our problem.

## 3.4 Pack

The implementation of Pack is highly dependant on the Scan implementation. This is due to the fact that this algorithm literally uses the Scan implementation as its main function.

There are three steps in the Pack pattern. Firstly we need to compute a bit-vector for true elements, which easly could be done with a parallel map. Secondly we execute a parallel fix-sum on the bit-vector, and finally we use parallel map to produce the output.

Of course all of this can be improved by combining the first two steps into one by changing the base case for the prefix sum and we can also combine the third step into the down pass of the prefix sum.

NOTE: Since the Pack algorithm is implemented with the Scan algorithm the same testing circuntances apply.

## 3.7 Pipeline

To implement the parallel version of the pipeline pattern, an array was created to keep track of the threadState (it saves a 0 or a 1, depending if the thread is free or occupied with a worker).

For a number of tasks, the data is copied and after that, I create a thread for each worker using a cilk\_spawn keyword. The spawn is first performed for one job, and while the worker is performing the calculations, this thread is considered occupied. After it is finished, the second thread is spawned, passing on this job to it, and so on, until the last job.

## 3.8 Task Farm

Implementation additional keywords:

workers – basic int[] array, containing information about workers, where 0 means that worker is free, and 1 occupied.

nWorkers – number of workers.

task – is function to be worked over data.

void passTaskToWorker(void \*dest, void \*src, void (\*task)(void \*v1, const void \*v2), int \*workers, int nWorkers){

int \*worker = getAvailableWorker(workers, nWorkers);

task(dest, src);

pthread\_mutex\_lock(&farmWorkerMutex);

\*worker = WORKER\_FREE;

pthread\_mutex\_unlock(&farmWorkerMutex);

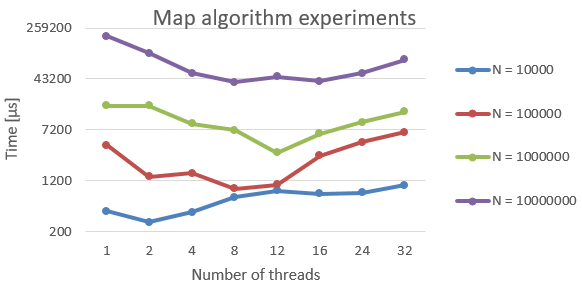
pthread\_cond\_signal(&isWorkerAvailable);

}

Above function is used by parallel loop. It gets first available worker. If there are no free workers, thread will wait for first one. getAvailableWorker function is also synchronized, since it access shared, worker, array. After task is finished by worker pthread\_cond\_signal is sent to indicate that worker is free (where isWorkerAvailable mutex\_cond is used inside getAvailableWorker function). Test functions were too small in comparison to overlay of getAvailableWorker function and mutex signaling hence usually only one worker was working. Much more complex tasks (or data heavy) are required for each worker too see benefits of parallelization.

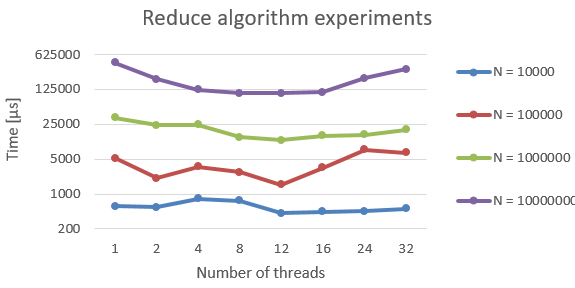
# 4 Experimental evaluation

## 4.1 Map



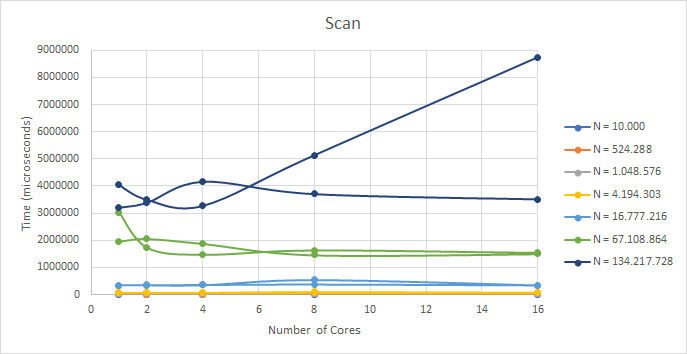
It is clearly visible that in case of smaller tasks (N = 10 000) overlay of parallelization was too significant to provide any speed up. But the bigger processed data, the more performance increase is visible. It is also clear that if more threads are used that physically CPU has, parallelized programs begin to slow.

## 4.2 Reduce



Same conclusions as in point 4.1

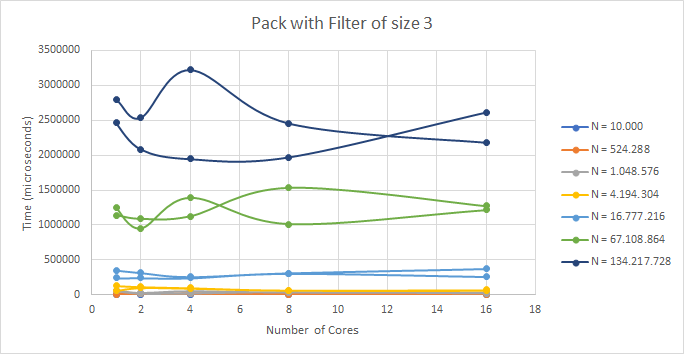
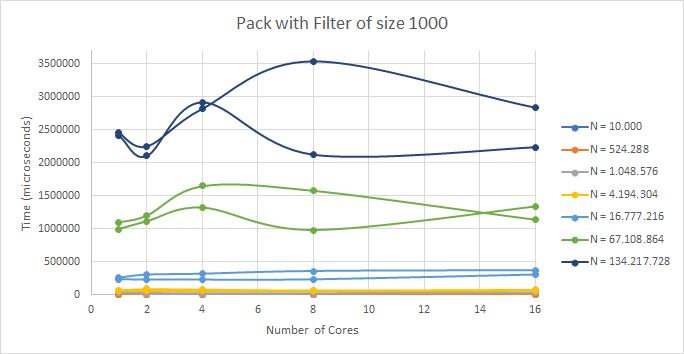
## 4.3 Scan

To test this pattern several tests were run where both the input length and the number of cores utilized in the process. More specifically, tests were run where the input length would vary between 10000 and 134217728 (only using lengths) and were tested in 1, 2, 4, 8 and 16 cores.

## 4.4 Pack

To test the Pack pattern the same tests as the Scan pattern were run, but this time it was also tested with a filter of length 3 and a filter of length 1000.

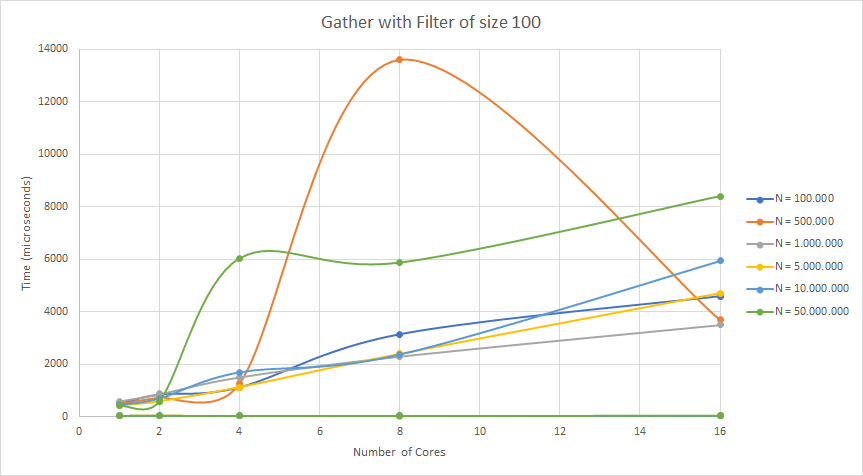
Here are the results:



## 4.5 Gather

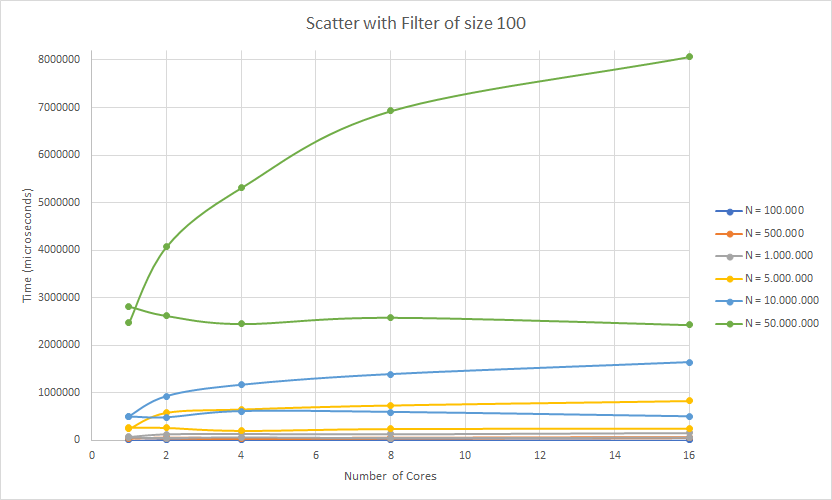
To test this pattern were run several tests where both the input length and the number of cores utilized in the process. Similar to the tests run on Scan and Pack. With the exception that these were run with a fixed filter of size 100 and the input length varied between 100.000, 500.000, 1.000.000, 5.000.000, 10.000.000 and 50.000.000.

The cores utilized also varied between 1, 2, 4, 8 and 12.

Here are the results:

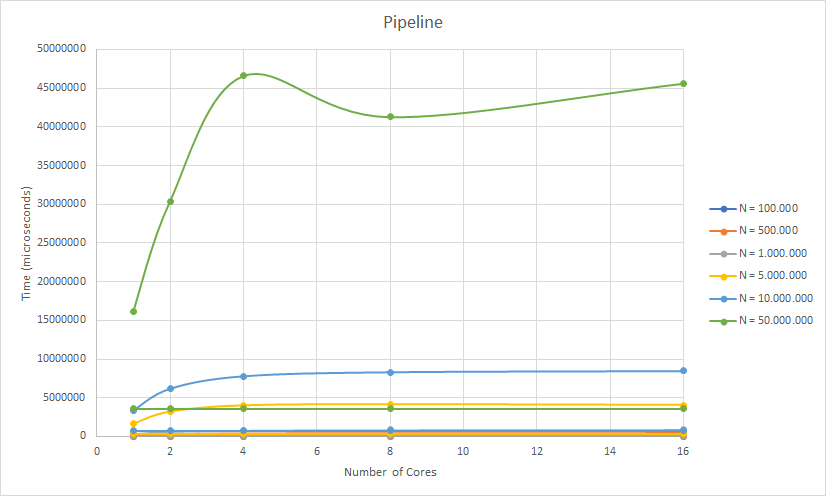
## 4.6 Scatter

To test this pattern the same tests from Gather were executed.

Here are the results:

## 4.7 Pipeline

Along side Scatter, the tests utilized in Gather were also executed on Pipeline.

Here are the results:

## 4.8 Task Farm

As it was expected, with more used threads program became slower due to getting free worker functionality overlay which was blocking threads. Parallel worker task calculations were too small to overcome overlay and show speed up.

# 5 Conclusions

Simple parallelization patterns (like map or reduce) are commonly used in most modern high level languages (python, JavaScript, Java, C#...), with some restrictions, because they provide vastly improvements regarding operating on collections, for example mapping and filtering huge data from database, with well-defined boundaries. It is given mainly because of their simplicity along with required task atomization. But more complex patterns like pipeline or task farm have to be deeply considered if their implementation will provide any benefits because of their more sophisticated purpose (like it is shown in our task farm implementation, where functionality regarding worker dispatching was a lot more time consuming than itself task to be done).

Also parallization tasks should be big enough to use threads, because usage of multiple threads yields additional callculations, so given task has to overcome it.

Also sometimes avoiding complex data structures can provide significant efficiency increase if execution time is most important (and it usually is, that is why tasks are parallelized, to reduce time executions).

Modern compilers are really effective in doing their job. To adjust compiled code, broad wide knowledge and comprehension of assembly language specifics are required.

**References**

1. Codeproject.com, ‘Writing Efficient C and C Code Optimization’. [Online]. Available: https://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization. [Accessed: 02-Dec- 2018].
2. cilkplus.org, ’Cilk Plus Tutorial’. [Online]. Available: https://www.cilkplus.org/cilk-plus-tutorial. [Accessed: 28-Nov- 2018].
3. McCool M., Arch M., Reinders J., Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann (2012), pp. 88-89.
4. McCool M., Arch M., Reinders J., Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann (2012), pp. 90-92.
5. Summerofhpc.prace-ri.eu, ’If not task farming then what’. [Online]. Available: https://summerofhpc.prace-ri.eu/if-not-task-farming-then-what/. [Accessed: 28-Nov- 2018].

**Comments**

1. Project in my opinion should be developed from the beginning to the end of semester (even of the „price” of being bigger or more time consuming) because students are here flooded with work in the end of November and that leads to much worse quality of final results. Our projects are usually counted as distinct course block along with labs and lecture (were these 3 blocks are counting as whole one subject for let’s say 6 ECTS points).
2. More emphasizing on labs and project instead of lecture and theory -> there is a simple reason of this. If we are about to practically do some work, we have to digest theory and apply it, and our understanding (and memorizing) is much better since we were able to “touch and feel” it. Most students forget about “dry” theory right after tests, because they never be able to practically work with it (and the only thing that we remember after hundreds of pages of dry theory is that we have read this a bit ago, and it was about <book domain>). Definitely the best laboratory session was the first one since it forced us to think and be creative. I really liked this. More of these will lead to better grades and more commitment. Also I wanted to mention that if project will be from beginning of semester, 1 tests grades would be a bit better (since they share scope).
3. Hardest part about this project was obviously C, and I see no reason to force us to use it. Most people spend most of the time digging in C specifics instead of focusing on parallel, threading, scalability, efficiency and foremost final report. For instance we should be able to choose preferred language (with exclusion to usage of concurrent collections for instance) because this project is not about showing shortest execution time but row of acceleration and this can be done using most of languages.  
   “*Be able to use the Java/C-like programming languages and parallel libraries to develop parallel software systems*;” this is extract from subject description from http://www.unl.pt/guia/2016/fct/UNLGI\_getUC?uc=11158. Well I haven’t seen introduction to Stream. Parallel, or ForkJoinTask classes even once. And to be honest I’m disappointed because of it a lot, that was the main reason why I enrolled to this subject, not another, to discover java parallel capabilities.
4. The thing am missing here is to do really cool parallel, multithreading program which would touch a lot of domain problems and concerns like I had as a part of Operating Systems subjects: we were given freedom in project proposal but it has to be multithreaded and somehow complex: the more threading ‘features’ or problems was it solving the better grade was, and so I did program based upon slopes and skiers, and this is why even after 2 years or even more I still remember about existence of mutexes, conditional variables, deadlocks and task starvation, not because of theory which I had to pass at lectures (a bit reference to point 2). If you wish to take a look into this program I still have it on my repository along with project description.
5. Also idea which are really liked was introduction to patterns since they are really important in programming, because just from a glimpse, everyone already knows what this part of program is doing but on contrary question on test about proper template of pthread\_create function it was not in place because its directly written in “Subject matter” section at online description of course, that we are not in C classes but in parallel and concurrency classes (obviously we have to write programs in some programming language, but test should regard characteristics of it). But overall implementing these patterns was also really cool because it allows us to take a look into very details of implementations, and this is really important to understand tools which we are using. (like it is in case of learning assembly language, after these classes I never had to google pointer tutorials anymore..) These were few cents from me, and I hope that they will contribute to subject improvements.