Optimizing a Software Introducing Parallelism: A Java Case Study

Vinícius Yamauchi – 28/07/2018

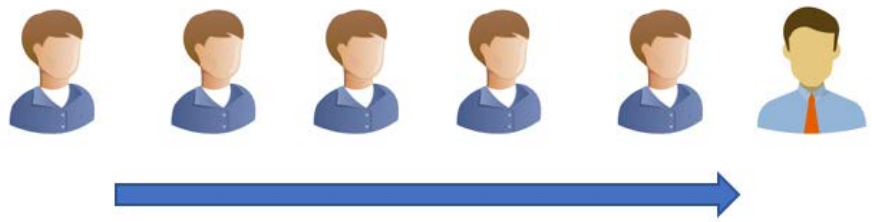
vs.souza@gmail.com

Recently, I was working on a software that was responsible for migrating content between databases and it was necessary to convert the data between two different NoSQL formats. The migration process used to take about 3 hours and we were trying to find a way to speed it up. After discussing with the team, we decided that we would work with parallel streams to see if we could get any benefits out of the box. For our surprise, the software performed worse after the changes. What used to take 3 hours to execute sequentially started to take approximately 3 hours and 15 minutes.

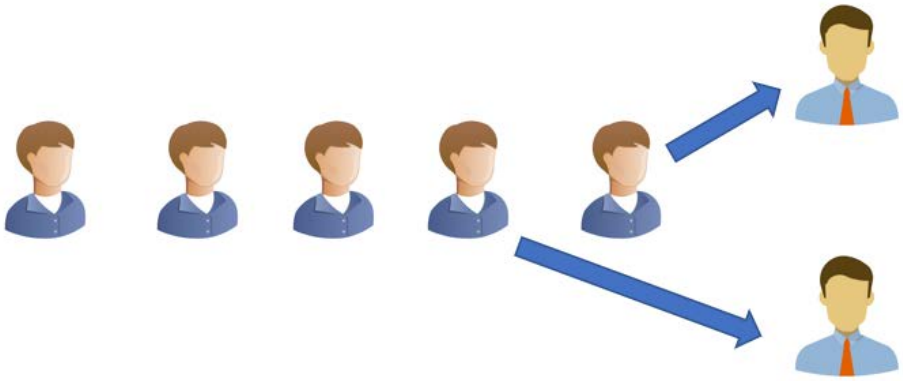
We didn't have too much time but I decided to reproduce here what we did, our mistakes, how we solved the problem and what would be the correct way if we wanted to work with Parallel Streams/divide and conquer. So, to reproduce what we had back there I've created a simplified scenario and we will walk through some parallel optimizations on it.

What does "to do things in Parallel" Means

To make it simple, doing things in parallel means to perform multiple tasks at the same time. Let's get an example on that. Imagine you enter a bank and there is just one bank teller available. There is a queue of 10 persons and, as imagined, the bank teller would have to handle the first person's request. After finishing this customer would leave and the next would be attended and so on. In this example, the bank operation is happening sequentially. Each customer is fully attended while the others wait.



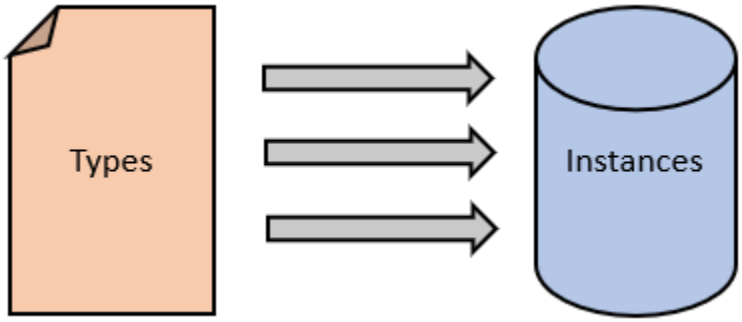
Now imagine that another bank teller got back from lunch and started working too. The next customer from the line got attended by the new teller while the other customer is still being handled by her team worker. So, from the bank operation point of view we have two customers served at the same time and that means we are performing multiple tasks at the same time. In this example we can identify that the queue would decrease much faster than if we had only one bank teller, right? Welcome to the world of parallelism!



I believe you already listened to this example before but this is the classic case of parallel execution in real life. I've started on computers in a remote era where we didn't have multiple cores in a processor. At that time, we had a strategy to emulate parallelism splitting the processor time between multiple small tasks of different activities. That was so fast that the user had the impression that things were happening at the same time, but actually we had the computer doing one stuff at a time. Today's computers have the capacity to really execute multiple tasks at the same time and that is what we will discuss, implement and analyze here.

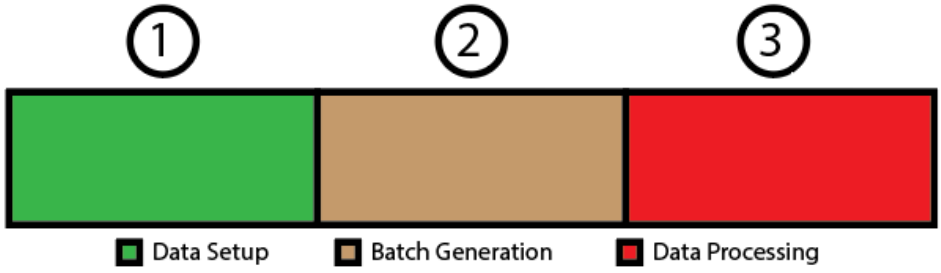
Establishing our Scenario

To demonstrate how parallelism can benefit your software we will create a practical example. For that we created a scenario where we had some elements in the database and we have to process them all separated by type. To make it clearer, basically we have types in a file and we have to find all instances of those types in the database and process them.



For each type we will retrieve all elements and do a simple update on the document. Replace "update on the document" for any computation you may imagine (like check for a set of constraints and based on that remove the document). To process sequentially we just need to retrieve the types, all instance IDs for each type and execute our computations based on the current instance type. To execute in parallel, we need to exercise how to divide the tasks properly and what is a good granularity for each task.

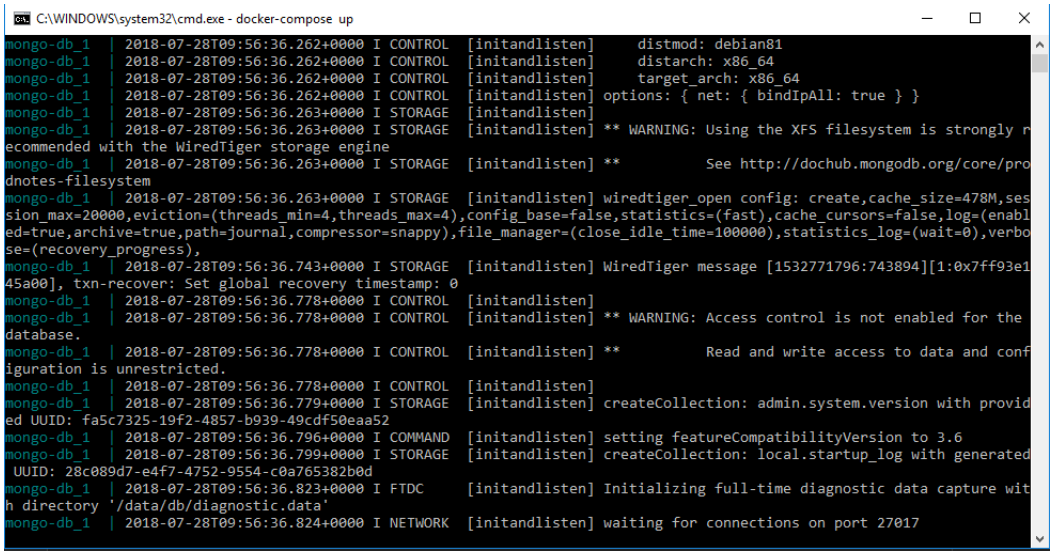
From a very high overview our scenario can be divided into three parts. We generate the data, we gather the IDs and we process the data. That is a very rough simplification but thinking like that will make it easier for us to attack problems in the optimization stages.



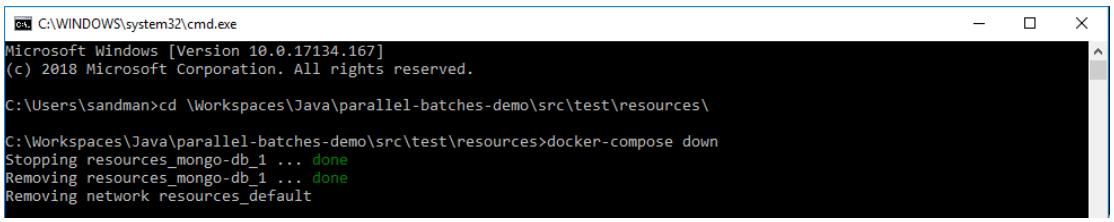
In the previous image we can see them in sequence. We first generate the data (1), then we gather the data to be processed (2) and finally we process the information (3). **Spoiler Alert… We are thinking sequentially!!!**

Bringing our Scenario to Reality

We need to choose the softwares we will be using for the project to implement our scenario. The choices were not based in any criteria (apart from the programming languages). To implement the previously described scenario we needed a database and the chosen one was MongoDB (any database would do the trick and MongoDB was not an exception in this case). The programming language is Java and we only use Java API except by the library to connect to MongoDB.



One important point to remember is that to stop the docker container properly you should open another console and navigate to the resources directory. From there you should run `docker-compose down`.



To build the solution we are going to use Maven. To run the database, we chose to create a docker-compose (that makes it easier for me to provide all the files you may need to run the project on your

own). I also chose JProfile to extract runtime metrics and IntelliJ as my IDE (you may not have any problems using a different IDE like Eclipse or NetBeans). In summary, our stack is:

- **Java 8+ (Required)**
- **MongoDB (Required)**
- **Docker (Required)**
- **Maven (Required)**
- **JProfile**
- **IntelliJ**
- **Robo 3T**

To simplify the project all output is gathered from console. All code is available in GitHub so if you want you can download the Zip file from **https://github.com/vssouza/parallel-batches-demo** (I would recommend you to install git and clone the project instead). I also made available the JProfile snapshots so if you want to take a closer look on the execution through time you may need JProfile 10.

The important sections of JProfile analysis will be available as image here and to download in the project. Don't worry about generating data for the project execution because the code we wrote already generate the file containing the types and the database elements that will be used.

The Runtime Hardware Specification

To execute this proof of concept we are using the following hardware specification:

- **Intel Core I7-2600 CPU @ 3.4GHz**
- **16 GB (4 x 4GB) RAM DDR 3 (1666MHz)**
- **256 GB SSD Sandisk**

The operational system in use is **Windows 10 Pro 64-bit**.

Building our Environment

Now I will presume you already have every software mentioned previously installed and configured. There are only three steps to get ready to run the project:

- *git clone* https://github.com/vssouza/parallel-batches-demo.git
- *open a terminal in parallel-batches-demo/.scripts/docker/*
- *docker-compose up*

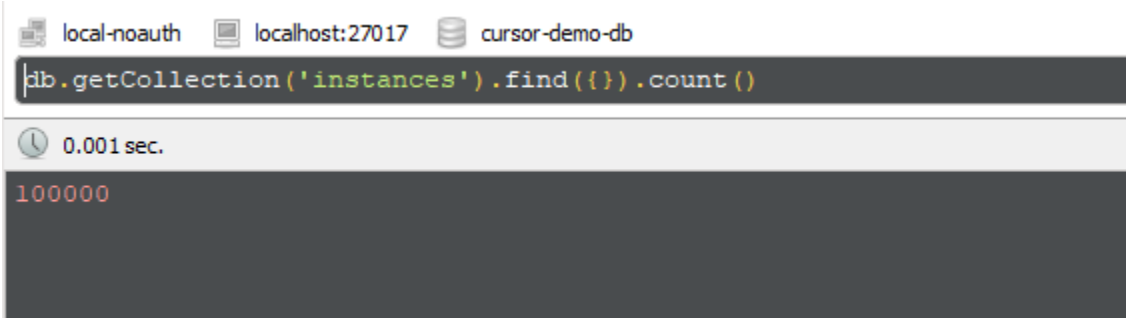
That's it! You have the database running and the code is properly deployed to your computer. The clone command will get the repository from GitHub and the docker-compose command will get the docker image containing MongoDB and run it exposing the correct ports. For this example, we are using MongoDB without authentication. Don't worry about creating the database structure as well because the Java code will do that in runtime.

If you want to profile the solution you have two options. One is to integrate JProfile into your IntelliJ, right click the main method and select Run with JProfile. Or you may open JProfile and choose to connect to a running VM in your machine. Remember to use sampling method to avoid too much overhead from code instrumentation (even that it is less precise).

Java Code Overview

The point is not to emulate a real software export but just to go through creating a software that executes tasks sequentially, analyze its problems and go through the optimizations. For that we

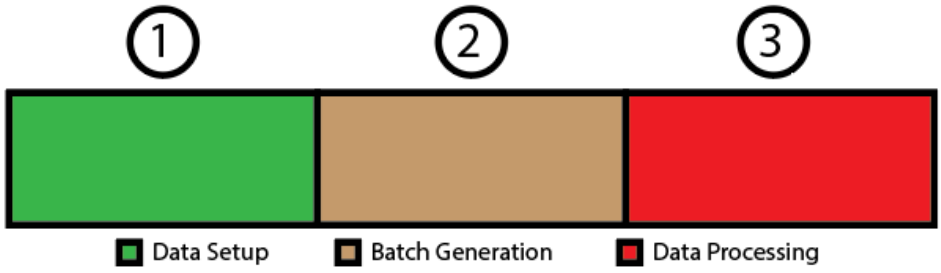
decided that our environment will have 10 types and each will have 10000 documents in the database. So, in total we will process 10000 documents.



We also created an Enumeration to define what type of execution we are going to perform (Sequential, Parallel Stream or Executor Service). It's good to keep in mind that Parallel Stream and Executor Service are both different strategies to run software in parallel.

```
1  package br.com.cursor.demo.executor;
2
3  public enum ExecutorType {
4      SEQUENTIAL, PARALLEL, PARALLEL_EXECUTOR_SERVICE
5  }
```

That makes it easier to execute the three scenarios we will be going through. The classes that implements each execution strategy are `SequentialExecutor`, `ParalllelExecutor`, ParallelServiceExecutor (I know… I know… the names are terrible but it was harder to name than to implement them). The class that contains the main method is `ParallelBatchesDemo` and this is the software's starting point. We'll also execute the data generation (block 1 from our sequence image) always in parallel in a separate block since it won't take a long time and doesn't matter for our analysis. We still perform it the same way for all the execution types to minimize possible interferences in the final outcome.



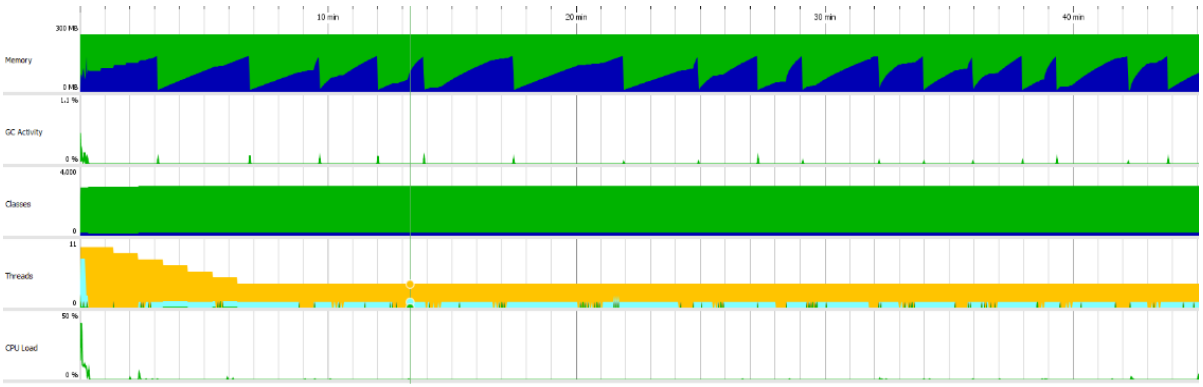
Apart from that it is important to mention that we created Singletons for the MongoDB database connection and also to generate Service Executor so they can be shared through the software. Those are expensive resources and we should reuse as much as possible to avoid waste of time and resources. It is also important to mention that the software cleans the database and file after each execution to properly get ready for a new execution without human intervention. If you stop the execution before it finishes you should clean the database and remove the file as well.

We also added some thread sleep on batches generation and batch processing. That would make some tasks take longer to run than others. It will show a possible problem when we use parallel

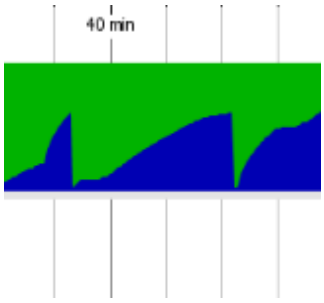
streams the wrong way. It also makes the software closer to real scenarios where not every activity takes the same amount of time.

The Sequential Execution Analysis

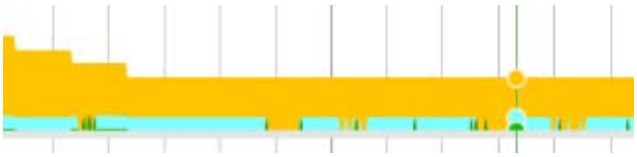
Well, looks like everything is ready so we will go through the first execution. Our first strategy will be running the software in sequential mode. This means we will first gather each type from the file one by one and then retrieve the ids for each type. The ids for each type are separated into groups of batches and then each batch is processed. While we are processing one batch the others will just wait for their time. In summary, we are executing the block 1 in parallel and after that every task is performed one at a time while the others just wait. Let's take a look at the execution overview:



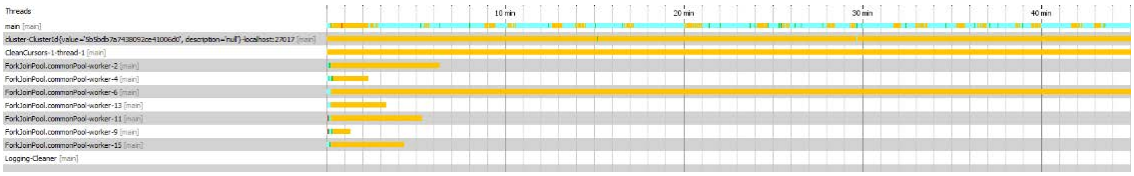
Yes, I know. It would be easier to find Ant Man in the quantic universe than reading something in the previous image. The first line shows the memory usage. The second the Garbage collector activity. The third the thread executions and the forth the CPU load (the full-sized image is available in the resources folder from the project so you can zoom in). Let's zoom in some important points and analyze it clearly. The first important point to check is that it took approximately 45 minutes to finish our execution. If you consider that we are just retrieving a bunch (to be more precise 100000) documents and inserting only an attribute that is a LOT of time.



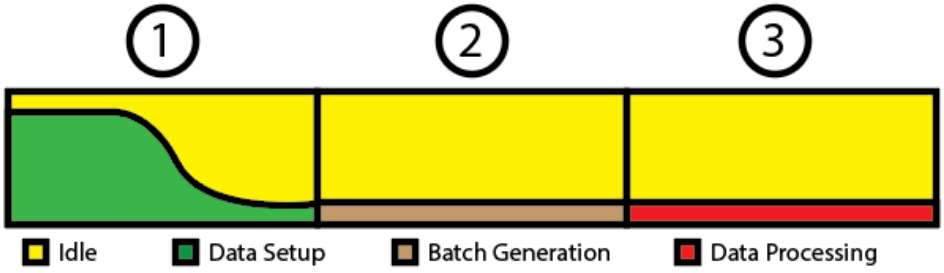
The second important point is that we can see that through all the execution our thread telemetry followed the same pattern. It has a huge yellow top and some small blue and green line on the bottom. The yellow top shows the idle threads. That means the threads that are there… just hanging around waiting for work that (in our sequential execution) will never come. That is a huge waste of resources since they already exist and are waiting to get a heavy load of work.



Our execution doesn't look to good in my point of view, but maybe the threads telemetry can prove me wrong. Let's check it out to understand how the threads behaved in our current execution (again, the full size image is available in the project resources folder):



From this we can see that there is just one thread (the first line) working through all our execution. That thread is the main thread. This just proves that we are wasting resources since there are other existent threads created that are idle all the time.



What we have is sketched in the image above. Just one thin line representing the active resources in blocks 2 and 3 and a lot of wasted resources represented as the yellow blocks. Just to remember we have a job setup (data retrieve + build tasks) on block 2 and batch processing on block 3.

We can see by the execution output that we first finish generating batches (in order) and after that we process data (following the batch job generation order).

```
…
Retrieving batch jobs for namespace0 type0.
Created 20 batch jobs for namespace0 type0 with a total of 10000 ids.
Finished retrieving batch jobs for namespace0 type0.
Retrieving batch jobs for namespace1 type1.
Created 20 batch jobs for namespace1 type1 with a total of 10000 ids.
Finished retrieving batch jobs for namespace1 type1.
Retrieving batch jobs for namespace2 type2.
Long running batch type retrieve activity.
Created 20 batch jobs for namespace2 type2 with a total of 10000 ids.
Finished retrieving batch jobs for namespace2 type2.
Retrieving batch jobs for namespace3 type3.
Created 20 batch jobs for namespace3 type3 with a total of 10000 ids.
Finished retrieving batch jobs for namespace3 type3.
Retrieving batch jobs for namespace4 type4.
Long running batch type retrieve activity.
…
Finished processing batch job 0 of size 500 for namespace namespace0 and type type0
Processing batch job 1 of size 500 for namespace namespace0 and type type0
Finished processing batch job 1 of size 500 for namespace namespace0 and type type0
```

```
Processing batch job 2 of size 500 for namespace namespace0 and type type0
Finished processing batch job 2 of size 500 for namespace namespace0 and type type0
Processing batch job 3 of size 500 for namespace namespace0 and type type0
Finished processing batch job 3 of size 500 for namespace namespace0 and type type0
Processing batch job 4 of size 500 for namespace namespace0 and type type0
Finished processing batch job 4 of size 500 for namespace namespace0 and type type0
Processing batch job 5 of size 500 for namespace namespace0 and type type0
Finished processing batch job 5 of size 500 for namespace namespace0 and type type0
Processing batch job 6 of size 500 for namespace namespace0 and type type0
Finished processing batch job 6 of size 500 for namespace namespace0 and type type0
…
Processing batch job 0 of size 500 for namespace namespace1 and type type1
Long running batch job id 0 for batch job 0 of size 500 for namespace namespace1 and
type type1
Long running batch job id 300 for batch job 0 of size 500 for namespace namespace1
and type type1
Long running batch job id 400 for batch job 0 of size 500 for namespace namespace1
and type type1
Long running batch job id 2100 for batch job 0 of size 500 for namespace namespace1
and type type1
Long running batch job id 2400 for batch job 0 of size 500 for namespace namespace1
and type type1
Long running batch job id 2700 for batch job 0 of size 500 for namespace namespace1
and type type1
Long running batch job id 3200 for batch job 0 of size 500 for namespace namespace1
and type type1
Finished processing batch job 0 of size 500 for namespace namespace1 and type type1
Processing batch job 1 of size 500 for namespace namespace1 and type type1
…
```

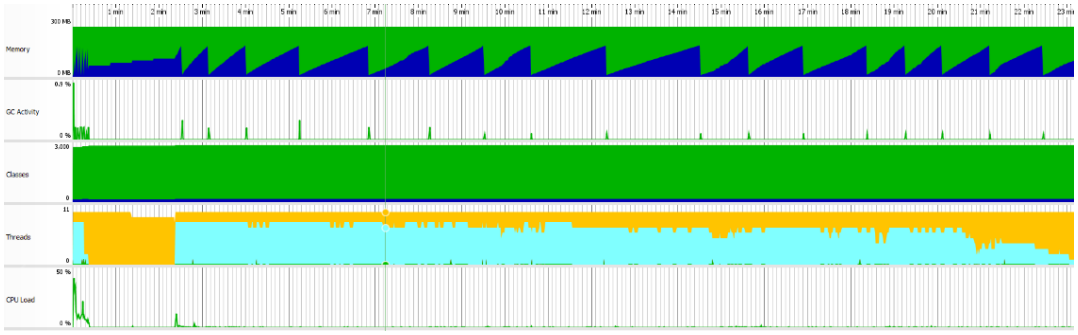
This is a sample from the console output for the execution (the file is available and you may consult the Appendix A to check the path).The saved output confirms that we have an execution order and that is achieved running every task using a single thread.

Introducing Parallel Streams

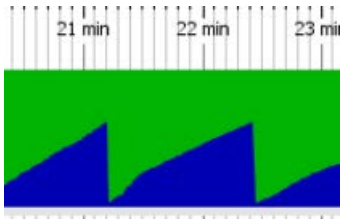
We just saw that we have a possibility to optimize our software doing things in parallel. Reading about the Java 8 APIs for parallel execution we found that maybe we can take advantage from the parallel stream feature. That makes simple to execute code in parallel and does not takes a lot of code. Other nice feature is that the syntax is really clean and straightforward to read (soooo cool)!!

```
int processedCount = batchJobs
        .parallelStream()
        .mapToInt(this::processBatch)
        .sum();
```

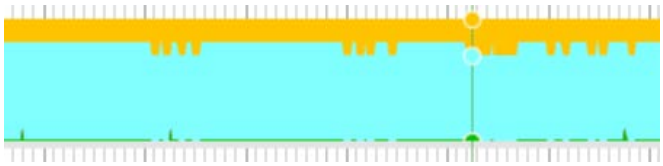
Let's take our last drawing and try to understand what we want to achieve applying parallel streams. Our target is obviously to use all resources available as long as our software is running and try to decrease the execution time (without getting our computer totally wasted out of resources)! After creating the class to introduce parallel streams on blocks 2 (data retrieve and batch generation) and 3 (batch processing) we have the following execution metrics:



There are three important points to check on the previous telemetry. The first one is that with this small change we have a slight increase of activity on GC metrics (second line of the image), let's park it from now since we will get back at this in the "Take Care on Some Points" topic. There are three major points to analyze here. The first one is that we drastically reduced our execution time. Our simulation now runs in approximately 25 minutes.

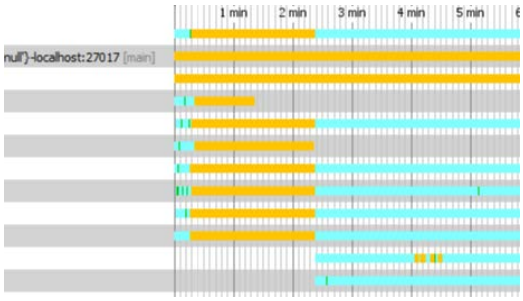


That is a 20 minutes reduction in our total execution time. We start to see the power of parallel execution after applying one simple change. The second important point is that now we have a lot of threads working even that they are on IO waiting (blue color) because my docker can't handle all of them as fast as they require to. That is something very positive because we wanted to use the most of the available resources in our software.

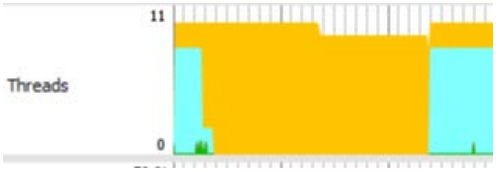


Let's take a look at the thread telemetry to check if our impressions get confirmed and we have more resources in use through time.

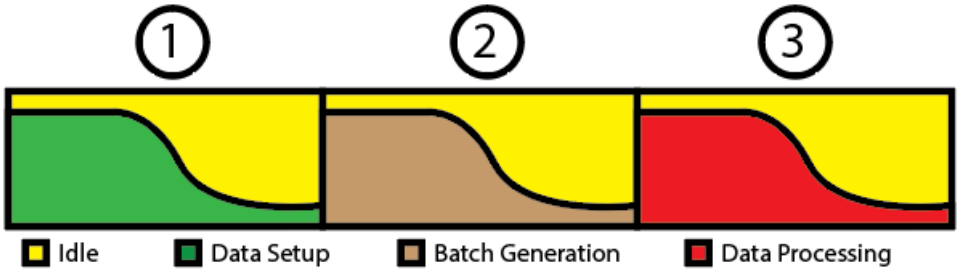
Wow! That looks much better. We have much more blue blocks going on and that means more threads working. Now we can see that at some point our main thread (first line) stops working and just wait for the job to be done by the other threads (workers) and that is ok. But looking at the thread telemetry from both images there is something that pops into my eyes. I guess you saw that too… yes… we have a big yellow block intersection! First let's check that on the detailed thread telemetry:



Now let's find this block in the thread line from the overview telemetry to check if it matches and try to understand what is happening. And here it is:



This happens when we are retrieving the instances and generating the batches. Do you remember that I mentioned that we introduced some thread sleeping to make some types generation to take longer than others? This is the result we get. We finish to execute the fast batch generation but we still have to wait for the slowest ones to finish. This is an effect of the (inefficient) way we applied the parallel stream API. Let's sketch what is happening through a drawing below:



We can see that we have a clear division in our software that determines when we have to start to retrieve instances, create the batch jobs (2) and process the data (3). This generates blocks of time where we still have lots of idle threads. We have to wait for block 2 to completely finish before we start to execute block 3. That makes us to wait for every type instance retrieving that takes longer (simulated with the thread sleep) even that we already have free resources and could start processing data from the batch jobs generated for the fast types.

It is important to remember that this technique is not appropriate to our scenario but may be very useful in other situations. One good example where those would be applied is on sharding/aggregation for distributed computing. Let's say that we have a set of data and we need to process and spread evenly through computer nodes in different geolocations. That would require a join point before delegating to the remote servers. In that case we would have a solution like the mentioned in the previous graph and it would be ok.

We have to say again that this is caused because we applied parallel stream in an inefficient way. The efficient way would be to create big task (read the file) and from each line generate smaller tasks (retrieve the ids and generate the batches) and for each batch generate an atomic task to process the data. That is what we name divide and conquer. We divide huge tasks into smaller at each stage of

execution. The tasks are always submitted to the same Execution service that does not distinguish from tasks and just delegates an enqueued task to the next available thread.

By the execution output we can see that we still process two distinguished blocks. One to generate the batches and another to process the data. But now we cannot trust the order in each block because we don't have a guarantee of finish order and task pickup.

```
…
Created 20 batch jobs for namespace7 type7 with a total of 10000 ids.
Finished retrieving batch jobs for namespace7 type7.
Retrieving batch jobs for namespace8 type8.
Long running batch type retrieve activity.
Created 20 batch jobs for namespace8 type8 with a total of 10000 ids.
Finished retrieving batch jobs for namespace8 type8.
Retrieving batch jobs for namespace9 type9.
Created 20 batch jobs for namespace9 type9 with a total of 10000 ids.
Finished retrieving batch jobs for namespace9 type9.
Processing batch job 11 of size 500 for namespace namespace6 and type type6
Processing batch job 2 of size 500 for namespace namespace8 and type type8
Processing batch job 3 of size 500 for namespace namespace7 and type type7
Processing batch job 1 of size 500 for namespace namespace9 and type type9
Processing batch job 2 of size 500 for namespace namespace3 and type type3
Processing batch job 12 of size 500 for namespace namespace5 and type type5
Processing batch job 16 of size 500 for namespace namespace7 and type type7
Processing batch job 8 of size 500 for namespace namespace8 and type type8
Finished processing batch job 1 of size 500 for namespace namespace9 and type type9
Processing batch job 2 of size 500 for namespace namespace9 and type type9
Finished processing batch job 2 of size 500 for namespace namespace3 and type type3
Processing batch job 3 of size 500 for namespace namespace3 and type type3
Finished processing batch job 2 of size 500 for namespace namespace8 and type type8
Processing batch job 3 of size 500 for namespace namespace8 and type type8
Finished processing batch job 3 of size 500 for namespace namespace7 and type type7
Processing batch job 4 of size 500 for namespace namespace7 and type type7
Finished processing batch job 2 of size 500 for namespace namespace9 and type type9
Processing batch job 3 of size 500 for namespace namespace9 and type type9
Finished processing batch job 3 of size 500 for namespace namespace3 and type type3
Processing batch job 4 of size 500 for namespace namespace3 and type type3
Finished processing batch job 3 of size 500 for namespace namespace8 and type type8
Processing batch job 4 of size 500 for namespace namespace8 and type type8
Finished processing batch job 3 of size 500 for namespace namespace9 and type type9
Processing batch job 4 of size 500 for namespace namespace9 and type type9
Finished processing batch job 4 of size 500 for namespace namespace7 and type type7
…
```

This is a sample from the console output for the execution (the file is available and you may consult the Appendix A to check the path). We still can trust that the blocks will be executed sequentially but we cannot guarantee that tasks within them will follow any sequence.

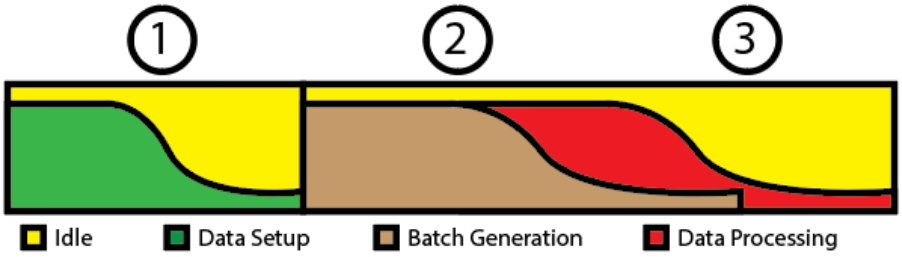
Changing to Executor Service

Now that we understand the problem and we have it clear how to solve it, let's create a new implementation that ignores types of tasks and eliminates that undesired barriers between block 2 and 3 and just executes tasks (doesn't matter their types) whenever a thread gets free. To do that we decided to create a Fixed Thread pool with the number of threads (workers) of the number of cores available in our PC.

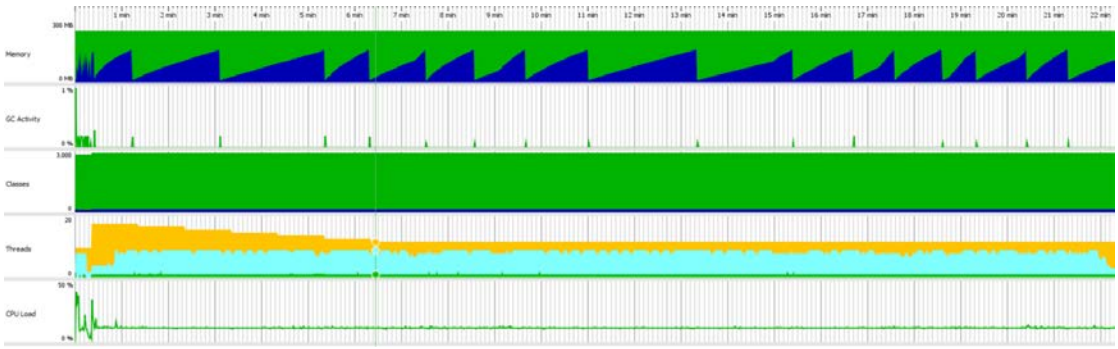
We also created two task definitions, one for the type 2 (id retrieval and batch generation) and another for the type 3 (data processing). The executor service does not make any distinction between them but that doesn't mean we don't have to execute different code for each block. The blocks still exist

but now there are no barriers separating their execution. We basically create tasks of data retrieval for every type in the file.

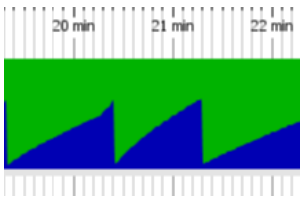
Each of those tasks are submitted to our Executor Service. Each of them generates one data processing task for each batch job and submits to the same Executor Service. This way we always have tasks enqueued until the software finishes. If one thread is idle it will get a task from the queue, that may be a batch generation or a data processing (for the thread it really doesn't matter at all). A graph representing the strategy we are adopting would be something like this:



On the previous image we can see that our target is to reuse the most resources possible between blocks 2 and 3. There is not a clear distinction between them anymore so they are merged in one big block that is processed by the same mechanism. Through that we start to execute the data processing block before we finish to execute the batch job generation block. Let's take a look at the overview telemetry for the new execution.



There are 3 important points to analyze in this telemetry. The first one is that we reduced the execution by approximately 2 minutes. It doesn't look too much but for a software that just updates 100000 documents on MongoDB this is a lot. And during optimization phase, if you are trying to decrease execution time every second counts. This example displays a small impact on the execution time but the benefits would be more visible in a CPU intensive case executed against a server with more cores available. On the other side if we take the sequential version of the same CPU intensive case to the same server described before there would be no changes at all.

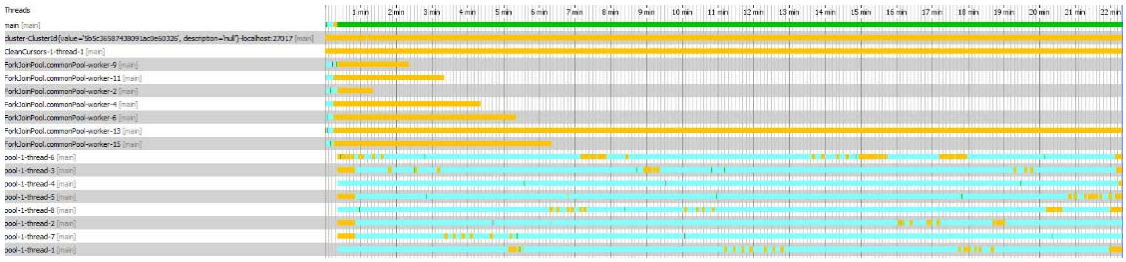


Another important point is that our software became more CPU intensive. You can check that on the last line of our overview telemetry that represents the CPU Load. That is not important in this

scenario because it doesn't reach 50% in the highest spikes but this is a change and we must consider in our analysis. The second important point is that we can see that the telemetry shows much more threads than the previous telemetries. That is because in the data generation we use the common fork join pool available (that generates threads) and for the blocks 2 and three we create a new fixed thread pool that gets stacked over the previous one. That is nothing critical too but that explains the changes in the graph.

The third and most important point is that we cannot see the idle intersection for all threads and there is always lots of resources in use. That was our main target with this change. Let's take a look in the thread telemetry and search for that idle block we identified before.

We can see that every thread is working where there was a lot of idle threads before. We did it! Let's check the entire thread telemetry to discuss a little bit about what is going on with our new implementation.



Here we can see that the main thread is always active and that is because now we have to check for service execution finished before continuing the main thread execution. If we do not create a join point we would close the database connection and clean the resources even before our threads get warmed up. That is not what we wanted, right? We controlled this using atomic integers to store running, queued and processed threads. Another option is to use Future (isDone method) object if your workers return any results (implements Callable instead of Runnable).

In the previous telemetry we can see as well the fork join pool used during data generation and then not used anymore. The same behavior can be identified in the fixed thread pool workers. They don't exist in the beginning of the execution but we create them to submit our batch generation tasks (and every task after that). We can confirm that through the execution output where we find tasks to generate batch mixed in between with tasks of data processing:

```
…
Created 20 batch jobs for namespace7 type7 with a total of 10000 ids.
Finished retrieving batch jobs for namespace7 type7.
Processing batch job 1 of size 500 for namespace namespace1 and type type1
Retrieving batch jobs for namespace3 type3.
Created 20 batch jobs for namespace3 type3 with a total of 10000 ids.
Finished retrieving batch jobs for namespace3 type3.
Processing batch job 2 of size 500 for namespace namespace1 and type type1
Retrieving batch jobs for namespace4 type4.
Long running batch type retrieve activity.
```

```
Retrieving batch jobs for namespace6 type6.
Long running batch type retrieve activity.
Retrieving batch jobs for namespace8 type8.
Long running batch type retrieve activity.
Retrieving batch jobs for namespace9 type9.
Created 20 batch jobs for namespace9 type9 with a total of 10000 ids.
Finished retrieving batch jobs for namespace9 type9.
Processing batch job 3 of size 500 for namespace namespace1 and type type1
Finished processing batch job 1 of size 500 for namespace namespace1 and type type1
Processing batch job 4 of size 500 for namespace namespace1 and type type1
Finished processing batch job 2 of size 500 for namespace namespace1 and type type1
Processing batch job 5 of size 500 for namespace namespace1 and type type1
Long running batch job id 2000 for batch job 0 of size 500 for namespace namespace1
and type type1
Finished processing batch job 3 of size 500 for namespace namespace1 and type type1
Processing batch job 6 of size 500 for namespace namespace1 and type type1
Finished processing batch job 4 of size 500 for namespace namespace1 and type type1
Processing batch job 7 of size 500 for namespace namespace1 and type type1
Long running batch job id 2300 for batch job 0 of size 500 for namespace namespace1
and type type1
Finished processing batch job 5 of size 500 for namespace namespace1 and type type1
Processing batch job 8 of size 500 for namespace namespace1 and type type1
Long running batch job id 3500 for batch job 0 of size 500 for namespace namespace1
and type type1
Finished processing batch job 6 of size 500 for namespace namespace1 and type type1
Processing batch job 9 of size 500 for namespace namespace1 and type type1
Finished processing batch job 0 of size 500 for namespace namespace1 and type type1
Processing batch job 10 of size 500 for namespace namespace1 and type type1
Finished processing batch job 7 of size 500 for namespace namespace1 and type type1
Processing batch job 11 of size 500 for namespace namespace1 and type type1
Long running batch job id 40400 for batch job 10 of size 500 for namespace
namespace1 and type type1
Finished processing batch job 8 of size 500 for namespace namespace1 and type type1
Processing batch job 12 of size 500 for namespace namespace1 and type type1
Created 20 batch jobs for namespace2 type2 with a total of 10000 ids.
Finished retrieving batch jobs for namespace2 type2.
Processing batch job 13 of size 500 for namespace namespace1 and type type1
Created 20 batch jobs for namespace4 type4 with a total of 10000 ids.
…
```

This is a sample from the console output for the execution (the file is available and you may consult the Appendix A to check the path). In this scenario we cannot guarantee any order of processing. No blocks or tasks order. That is important to consider when designing parallel solutions. The higher the autonomy of tasks the faster you can run.

Take Care on Some Points

When programming is important to remember that there are no silver bullets. Sequential execution is still important and has its place in the Computer Science. Due to our tendency to think in an ordered way we tend to not consider parallelism at all or even to implement half of a parallel solution like we did on the second step.

You also should take care on Full Garbage Collection when running lots of codes in parallel. If you load a lot of objects into memory per thread and they take a long time to run (not well dimensioned) you may end up having a lot of objects. Full GCs may freeze your entire application execution and that may lead you to run slower than before you applied parallelism.

Another important point is that if your problem demands a sequence to be solved you may not parallelize it. Thread creation is an expensive resource and maybe sometimes it may not worth the cost-benefit for your implementation.

Conclusion

If you apply parallelism correctly you may end up decreasing drastically the time your software takes to execute. We could see that in practice here through a concrete implementation. By experience I see a lot of people neglecting parallelism because they are afraid to deal with threads or are not used to design parallel software.

There are some trade-offs on using parallelism and one huge problem is that parallel software is much complicated to debug or detect problems. We enter some dark regions of race conditions, resources extinguish and deadlocks but the challenges worth the benefits for sure. Java made a very good job simplifying their API for concurrency and I believe that this may help to increase the number of developers that get used to think in parallelism.

Appendix A

All resources are available in GitHub. Once you download or clone the project you may find any file necessary for a more detailed analysis:

Source Code and Files: *https://github.com/vssouza/parallel-batches-demo*

Sequential execution folder: *parallel-batches-demo/.extras/java/runtime-sequential-execution*

Parallel Stream execution folder: *parallel-batches-demo/.extras/java/runtime-parallel-stream*

Parallel Stream execution folder: *parallel-batches-demo/.extras/java/runtime-executor-service*

MongoDB Images: *parallel-batches-demo/.extras/monbodb*

MongoDB Docker Compose: *parallel-batches-demo/.scripts/docker*

Each execution folder also contains a JPS file with all information collected during execution, the console output saved in a text file and two html files (one containing the overview telemetry and the other the thread telemetry from JProfile). A copy of this article may also be found inside the folder *parallel-batches-demo/.extras/*.

Revision Table

Name	Date	Comments
Marcus Rodrigues Eltscheminov	29/07/2018	• Suggested scenario where parallel stream model would be desired; • Fixed lots of typos and mistakes; • Suggested new structure to add the docker-compose file;