Bachelor Thesis Report

Release alpha

Anthony Hermans, Sam Mylle, Stijn Janssens, Federico Quin, Eve

Jun 25, 2017

Table of contents

CHAPTER 1

Group Apollo

Introduction

This is the report for our Bachelor Thesis project, which consists of extending the Stride software project.

Artifacts

To keep the main project clean of unrelated commits, we've splitted our work up into multiple repositories.

- Stride: Github, Bitbucket
  - User manual: readthedocs, pdf
  - Doxygen Reference Manual: readthedocs
- This report: readthedocs, pdf, source (Github)
- Prototypes
  - Checkpointing (HDF5): Github
  - Multi Region (MPI): Github
  - Unified Parallelisation (TBB): Github
- Website: apollo-stride.github.io, source (Github)

Other useful links:

- Travis CI

Functionals

Here we'll discuss our progress for each functional requirement.

Population generator

We had a rough start, since there was an inconsistency in the assignment. We wrote quite some code for this that was eventually not needed, since the assignment was changed.

After the change, we made reasonable progress and were able to implement all of the requested features. We planned the development of the population generator to be the first thing we did, so that we could use this generator in our tests and further development. We finished quite early in the project on schedule and were able to use it during the development of multi region.

Checkpointing

Checkpointing was handled in different stages. The first stage was checkpointing the already existing functionality of Stride. This was a clearly seperatable task and we started working on this right away. As HDF5 is a very scientific and broad language, the initial steps were difficult and came with lots of segfaults. However once we got the hang of it the progress of the checkpointing feature went rather smooth. Apart from a few minor adjustments due to constructive criticism, this stage was finished at the alpha demo. The second stage involved handling our own extra features with checkpointing, the major point of struggle was multi region. We had to save all the information of all the different simulators in an efficient way. We also needed to be able to restart a configuration the way it was before. This was a non-trivial matter and we communicated a lot with the different subgroups in the project. After a complete rewrite of the configuration system, which was necessary both for us and for the multi region / mpi tasks this part of the assignment was also finished succesfully.

Last but not least we had to make a trade-off between workload and user experience. It is not possible to restart an entire multi region simulation straight from only one configuration file, more specific we do not save an entire simulation to one file. Therefore the user can restart such a simulation in the exact way it was before, but only using the original configuration file, all other restart options such as timesteps are still available.

ParaView

We considered the ParaView task as a subtask of checkpointing and therefore it's a task for the checkpointing developers. We had trouble with ParaView, from start to end. We started working on ParaView in the middle of the first stage of the checkpointing assignment as we discussed, but we all had trouble compiling or running ParaView let alone using a plugin or writing one ourselves. This in combination with the duplicate way of visualizing using both ParaView and our own visualization tool, led to a tradeoff between functionality for the user and workload for the developers and resulted in us dropping the integration of ParaView.

Multi region

We splitted this in two parts, as described in the assignment. Our goal here was to make an interface that is designed in such a way, that you wouldn't notice the difference between the shared memory approach and the MPI approach. This had some serious consequences for the shared memory solution.

The first version was a shared-memory implementation. We had to rewrite the above mentioned interface 3 times. This was due to some misunderstandings and miscommunications. We spent lots of time on the design of a good configuration structure, as mentioned before. Once this was decided, the interface of multi region needed a rewrite but the overall development was easier.

As is discussed in our test plan, we were planning to create a unified interface for both our shared-memory implementation and our MPI implementation. This interface, which we called `AsyncSimulator`, has been a point of heavy discussion. It makes extensive use of `std::future` to provide the needed parallelism.

Distributed multi region

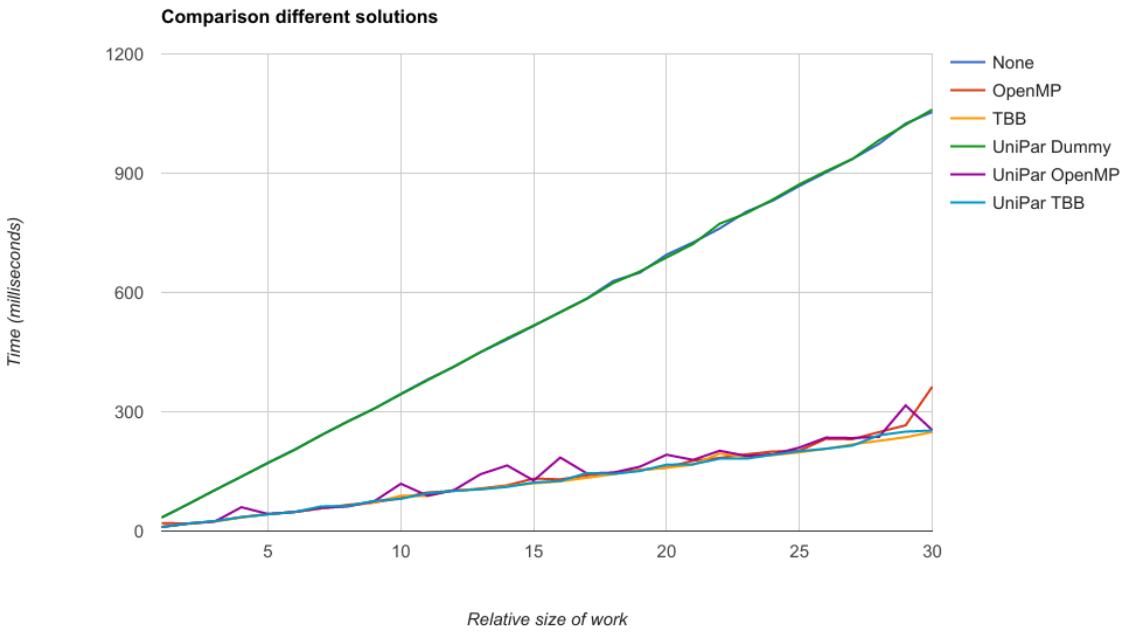
We made quite some progress in providing the necessary functionality for MPI, however we did not finish it before the deadline. Our experience with MPI did learn us that it may not be the best fit for this problem. In fact, a library like Avro (the RPC part of it – not available for C++) may be more suited for this. Strong serialization support (without extra code), as is standard in Java, would also be a huge help.

Because of our design, it is trivial to support mixing local and distributed simulators.

TBB (UniPar)

First, we made a prototype to familiarize ourselves with both OpenMP and TBB. The prototype also showed a small performance increase in favour of TBB. However, a second prototype that was more in line with how OpenMP was used in the main project showed equal performance for both OpenMP and TBB.

We decided to first develop our code for unifying both parallelisation libraries in the prototype. At that point, we also devised a name for our little library: UniPar, short for Unified Parallelisation. It makes extensive use of templates (including parameter packs) and lambda's (including `std::forward`). In our prototype, performance was identical between the different versions. (The entire table can be found here.)



Then we placed the UniPar library in the main Stride project. This did bring up some issues. The biggest one is the fact that TBB is a bit more high-level than OpenMP. The loop that used OpenMP in Stride, made use of an `RngHandler`. Those are kept in a vector, one for each thread. This vector is then indexed using `omp_get_thread_num()`. However, in TBB this function was (as advised by the OpenMP group at Intel) purposefully removed. This gave us two options; either provide our own way of getting a thread number in TBB (which would be against TBB's way of working), or devise another way of providing random numbers.

We went with the second option. A `ResourceManager` class was added to UniPar (not directly visible for clients). This uses `std::function` to provide the requested resource on demand. However, this meant we had to rewrite some stuff in Stride. After multiple tries, we settled on a method without `RngHandler`, using `Random` directly.

A new problem showed: although our "Dummy" implementation (without parallelisation) passed all tests, our TBB and OpemMP versions did not. The reason for this is that the previous OpenMP implementation, even though multi-threading is inherently nondeterministic, it did however provide "enough" determinism for much narrower limits on the results (i.e. `getInfectedCount`).

To verify that our implementation wasn't at fault, we tested the original OpenMP implementation with the `default` configuration (taken from the tests) on 500 random seeds generated by `std::random_device` (very much like our own UniPar implementation). The results were as follows (for day 30):

- Average: 78187
- Standard deviation: 5983
- Minimum: 56116 (70000 - 56116 = 13884)
- Maximum: 108126 (108126 - 70000 = 38126)

This shows us that when using random seeds, we are better off using an expected number of cases of 80000 with a tolerated difference of 30000.

Scientific visualization

We have chosen for a lot of technologies to help build our visualization tool. We use Electron to build a native cross-platform desktop application with web technologies like Javascript and CSS. Electron is able of extracting a buildable executable tool which is what we want for our visualization tool. Due to usage of web technologies we have access to a number of useful libraries. We use Mapbox for the mapview, Plotly for elegant graphs, AngularJS for the dynamic content of our tool and Material Design Lite for the overall styling. Most of these tools were used by at least one of the developers in this subgroup already, so the learning curve was not as steep as it might have been.

The development of the tool is finished, however the layout and design can always be improved, but that's not our top priority of course.

The first presentation of this tool was shown at the beta release, the overall feedback was very positive and the idea of the tool was right.

A few adjustments however needed to be done: We added aggregation of different cluster types onto the same location for more information in a less cluttered way. We also added a better overview panel complete with visualized data of our population generator. This last addition guarantees a nice cohesion between different parts of the overall assignment. We know also have an exciting airport function that shows airport travel and the area of influence.

Non-functionals

Coding style

We've defined a relatively strict coding style, and have committed to also convert existing code and new changes to this format.

macOS support

We initially had lots of troubles for macOS, mostly related to the ability of specifying the random generator(static and non-static min/max methods). But these problems are now solved. We no longer use polymorphism for the random generator. Instead we use a templated attribute of the population generator to specify the random generator. We chose to use the standard random generators (random library) because of the extensive documentation. Moreover the random generator is no longer specified in the XML config file. We now use the commandline interface to indicate which specific seed and which random generator you wan't to use.

Besides the random generators, there was also a minor issue with the visualization app. This problem was caused by a different folder structure of the Electron app on mac OS. The folder issue was resolved quickly and doesn't require further information.

Continous Integration

At first, we misunderstood the requirements related to CI. These requirements are relatively easy to do in Jenkins, but our Git Workflow (which we're very fond of) is thightly integrated with Travis.

As a solution, we've properly divided the build and run steps, and provided a much cleaner job log that gives you an instant overview of what tests failed.

We also use Travis Build Stages, a new beta-feature in Travis that allows us to build in 3 stages:

- Code quality check
- Unit Tests
- Scenario Tests

and stop a build as soon as one of the stages fails, that way we have faster cycles and a better workflow.

Documentation

We host the documentation, for both the report and the user manual, on readthedocs.io. Since our documentation will rarely get printed, it makes sense to focus on web first, and pdf second. However, readthedocs.io still provides pdf's.

We also make Doxygen documentation available through ReadTheDocs. We've tried integrating it into Sphinx, but the results were disappointing. For this reason, we simply used the html generated by Doxygen. (This too, is saved per branch).

Test plan

We use Google Test for our tests. We split the tests in two major parts: unit tests and scenario tests. We choose between the two parts via a parameter provided at runtime.

We have also decided that CTest did not have any more value to us, except making the build process more complicated. Therefore, we no longer use it.

Continuous Integration

There are 4 different choices to be made:

- (short) **unit tests** or (long) **scenario tests**
- **gcc** or **clang**
- **TBB**, **OpenMP** or the **dummy** implementation
- with or without **MPI** (however, see *MultiRegio*)

All tests are executed often; on all commits. A pull request may only be merged if CI says it's ok, on top of a manual review. To get quick feedback from unit tests, we split unit tests from scenario tests using Travis Stages, a very recent feature.

Assignments

For each assignment we'll discuss how we are going to test this functionality.

HDF5 checkpointing

We test checkpointing in different ways:

A first form of testing is checking the format and content of different HDF5 files. We make a difference between happy-day scenario files and wrong files (not according to the format, differing from the simulator's state, ...). Another form is running from various interim checkpoints, after which we compare the results from those runs with runs that ran uninterrupted from start to finish (when using no parallelization, this has to be an exact match).

TBB Parallelization

To test our implementation of TBB, we'll first write some scenario's and run them manually. After this, we check the results ourselves. If the results are free of errors, we create a test case from it.

Apart from that, we'll also write some unit tests to ensure the behaviour of the code of is as we expect in all three implementations (TBB, OpenMP, Dummy).

Population generator

There are a few classes that help the generator, for example one that provides an Alias distribution or a class that works with geo-coordinates. These classes will have unit tests for various edge cases and configurations.

Testing the generator itself is split in three parts:

- **Input files**: When input files are syntactically or semantically incorrect, the generator has to properly handle them.
- **Distributions**: We have to check whether the distribution of the generated data is as we expect. Extreme values shouldn't occur in sufficiently large populations. For example, it is virtually impossible for a random generated population of one million people to have half a million unemployed people when the given unemployment rate is 10%.
- **Output**: Parts of the output can be checked without resorting to statistical analysis. For example, all possible family compositions are known at the start. It shouldn't happen that a family is generated of which the composition is nonexistent.

Multi region

In our architecture we plan to abstract all MPI communication details in an interface equal to the one used in a shared-memory implementation. Once the MPI implementation for MPI is finished, we'll extensively test it. However, because of the added complexity that is distributed testing, we will do this manually and preferably only once. Testing with multiple processes (and not multiple machines) could be repeated more often. In the build configuration, MPI can still be turned on or off, but no unit tests will be written testing the actual distributed nature of the system. Therefore, the only reason to try compiling with MPI is to test whether it interferes with other components.

To write unit tests for multi region, we'll use only the shared-memory implementation. In theory, if the interface behaves exactly like the one used by MPI, this should be enough. To make up for the loss in testing, we plan to spend more attention to code reviews regarding MPI.

Scientific Visualisation

Scientific visualisation is primarily a GUI implemenation and therefore rather hard to test. The quality of our visualisation is also not objectively measurable. Therefore, we'll mainly test the scientific visualisation manually, throughout the project.

Documentation plan

Code documentation

We chose to have a not too strict form of documentation. Everyone is responsible for the documentation of his own code/functionality. Documenting the code of another person is discouraged. Also documentation that doesn't add additional information is left behind (see the following example):

```
/// gets age of Person
int getAge() const;
```

For reference documentation we chose to use Doxygen. This means we also use the associated syntax to provide more information for the arguments etc. We try to use the currently present documentation style as much as possible.

Hosting

We plan to use readthedocs.org for both the general documentation (manual) and the reference documentation (doxygen). ReadTheDocs provides a couple of handy benefits:

- Builds the documentation itself
- Available for every branch (=> works well with our git workflow)
- Easily searchable
- PDF support

User manual

Concerning the user manual, we make further use of the existing structure of the Stride manual. Each group will work on a feature. If the user manual needs to change as a result of such a feature the whole group will be responsible for this change. How this is arranged within the group is not determined. The existing chapters will be expanded/changed if necessary. The command line arguments for checkpointing (chapter 3.2 - Run the simulator) will be extended with the checkpointing frequency.

Next to the existing chapters we will add one more chapter. This chapter will provide more information about the newly added tools. For each tool we will write different subsections like the following:

- How to use the tool
- How to configure the tool in the project
- What is the function of the tool

For example if the user wants to use the multiregion extension he needs to know that it's possible that he is working with a distributed system. Below is a short general listing (per feature) of user questions and their specific answers:

- Checkpointing
  - How do I use checkpointing? How do I activate checkpointing?
- TBB parallelisation

  - How do I use TBB? How do I activate it?
  - What are the differences in performance in comparison to other solutions?
- Population generator
  - Where do I find the executable?
  - How do I use this generator? What is the correct XML format?
- Multi region
  - How do I use multi region? Are there consequences I need to be aware of?
- Scientific visualisation
  - How do I use this? What do I get to see?
  - What is the meaning of the results?