Project Report: Flu++ Group

Sibert Aerts, Cédric De Haes,
Jonathan Van der Cruysse, Lynn Van Hauwe

June 2017

Preface

This document is a retrospective report for our bachelor's degree group project: a fork of the *Stride* disease modeling simulator.[1] We will give an overview of the features we added, and describe the hurdles and obstacles we ran into while implementing each of them.

Contents

[1] Flu++ Group. *flu-plus-plus bachelorproef repository on GitHub*. 2017. URL: `https://github.com/flu-plus-plus/bachelorproef/` (visited on 06/16/2017).

1 HDF5 checkpointing

The HDF5 checkpointing was only partly implemented. There is also one additional feature: if an interrupt is signaled during a single-region simulation, the current step will be completed and the result will be saved.

1.1 HDF5 library

For this project, we used the recommended C language API from HDF5. If CMake detects the absence of the HDF5 library, the code relating to checkpointing is ignored during compilation. This behavior can be enforced using the `STRIDE_FORCE_NO_HDF5` flag.

1.2 Approach

We started with making a checkpointing mechanism for a single simulation.

We save everything used in the configuration. The files are stored as a string of characters. Other configuration variables are stored as attributes of the group containing those files. The `Atlas` and Town structures used by the visualization tool are also saved in this config map.

Other maps are named by their checkpoint date. These contain a dataset with all the info about the population, a dataset with all the extra information about the visitors, a dataset with all the information about the expatriates, and finally a dataset for each cluster type. A cluster is stored as a list of records containing a cluster ID and a person ID. The fictitious person ID 0 is listed for every empty cluster, to make sure such a cluster is represented by at least one record.

For a multi-region simulation checkpoint, we simply group all the single-region checkpoints from the individual simulations together.

1.3 Editing HDF5 files

We used the `hdfview` tool to examine the exported `.h5` files. We also wrote the `stridecp` program to be able to edit some parts of such a file. These parameters of a Stride `.h5` file can be tweaked using `stridecp`:

- The disease file
- The contact matrix
- Several configuration features:
  - Whether or not to track the index case
  - The R_0 value
  - The RNG seed
  - The amount of days to simulate
  - The logging flag
  - The prefix for the log files

1.4 Missing Features

There are two missing features in our checkpointing implementation: it does not interface with MPI, and we did not write a ParaView plug-in.

1.4.1 MPI

The whole MPI part is missing from this project. That is why checkpointing also does not have a MPI implementation. We did look for a way to do this, but we could not find an easy way to send data over an MPI link.

1.4.2 ParaView plugin

We heard from other groups that there were a lot of problems with ParaView (ParaView not compiling, segmentation fault on loading a plugin, . . .). This, combined with time pressure, explains why this feature is missing.

Cédric wrote everything to do with checkpointing.

2 Parallelization

The parallelization sub-task was implemented in full and extended with some extra features. Additional features include: a uniform parallelization

API, two parallelization implementations that do not depend on external libraries, and a parallel map implementation that has been integrated into the `Population` type.

2.1 Parallelization libraries

The CMake script will pick an appropriate parallelization library at configure-time and build Stride for that library.

We implemented parallelization by first creating a small common API (`src/main/cpp/util/Parallel.h`) and then implementing that API for specific parallelization libraries. In total we have created four implementations: OpenMP, TBB, a home-grown implementation based on standard library threads, and a reference implementation that does not distribute work across processing units.

The user manual includes a detailed overview of how the configuration process works, along with a performance comparison of parallelization libraries.

2.2 Parallelizing maps

2.2.1 Approach

We also tried to parallelize operations on (dense) `map<K, V>` contents. In this context it proved somewhat difficult to carve up the data structure in chunks before processing each chunk separately. `map<K, V>` does not provide an API that allows us to create such chunks nor does it support getting an iterator to the nth element like an `vector<T>`.

We resorted to a clever trick: we request the minimum and maximum keys in the `map<K, V>` and partition $[\min, \max]$ into n ranges $[k_i, k_{i+1})$ of (quasi-)equal length. We then request iterators to the key-value pairs with key k_i.

If some k_i is not a key in the `map<K, V>`, then we insert a dummy pair $(k_i, \texttt{V()})$ into the map, get the iterator to that pair, advance said iterator to the next pair, and delete the dummy pair we inserted.

Once we have an iterator starting at key k'_i for each k_i such that $k_i \leq k'_i \leq k'_{i+1}$ for $i \in \{0, 1, \ldots, n\}$, we spawn a number of threads an give each thread a

range of keys $\left[k'_i, k'_{i+1}\right)$ to process.

2.2.2 Complexity analysis

A simple complexity analysis shows that our chunking techniques scales well. Let $d(m)$ be the number of operations it takes for `map<K, V>` to get an iterator to an element, insert an element, increment an iterator, and delete an element. We know that $d \in O(\log m)$.

We want to start n threads, which implies that we need to obtain $n + 1$ iterators. This yields us an overall time complexity of $O(n \cdot \log m)$.

The complexity obtained above is quite suitable for quickly dividing a container into chunks—n tends to be rather small ($n \leq 16$ even for high-end desktop machines) and $\log m$ is also small, even for large values of m.

2.2.3 Drawbacks

There are two main drawbacks to the approach we use to carve up `map<K, V>` instances:

- We assume that keys are distributed more or less uniformly. This is a fundamental limitation: there's nothing we can do about. Fortunately, our assumption turned out to be mostly correct for all `map<K, V>` values on which we wanted to operate in parallel.
- Carving the `map<K, V>` into chunks mutates the container's contents. This is non-obvious and may bite users that want to run multiple parallel queries *simultaneously*—even pure parallel queries are thread-unsafe if the approach outlined above is used naïvely.

We mitigated the effects of the latter drawback by creating a new data structure: `ParallelMap<K, V>` wraps an `map<K, V>` and a reader–writer lock. `ParallelMap<K, V>` offers the same interface as `map<K, V>` and augments that interface with parallel and serial iteration functionality.

When an operation is to be applied in parallel to the contents of a `ParallelMap<K, V>`, a (unique) writer lock is held while chunks are created. All other operations, including the operations inherited from `map<K, V>`, acquire a (shared) reader lock. The consequence is that, if used in isolation, serial map functionality will never block, and parallel usage will block only while carving out chunks (which is rather fast).

Note that `ParallelMap<K, V>` is *not* a thread-safe wrapper around `map<K, V>`. Rather, it offers the *same* thread-safety characteristics, with the addition of thread-safe parallel and serial iteration functionality.

2.2.4 Performance analysis

Figure 1 presents statistics to evaluate iterating over all elements in a `ParallelMap<K, V>` serially or in parallel. Over the course of fifty runs per map size, eight threads were applied to `ParallelMap<K, V>` that contained one hundred, five hundred or one thousand elements. Processing a single element from the array takes approximately one millisecond. The measurements were produced by an Intel Core i7-6700K CPU running Ubuntu 17.04 and gcc 6.3.0.

elems	serial mean	parallel mean	serial stddev.	parallel stddev.	speedup
100	0.110s	0.028s	0.002s	0.007s	3.93
500	0.553s	0.083s	0.006s	0.007s	6.66
1,000	1.111s	0.158s	0.015s	0.010s	7.03

Figure 1: Statistics for evaluating `ParallelMap<K, V>` performance

The table above shows that `ParallelMap<K, V>` is quite effective for its intended purpose: processing map elements in parallel results in a significant speedup. Most importantly, the speedup factor increases with the number of elements in the map, which makes `ParallelMap<K, V>` quite suitable for the large collections managed by Stride.

Jonathan implemented parallelization.

3 Synthetic population generation

The population generation sub-task was fully implemented: if Stride is run with a population model XML file as the `population_file`, the parameters in that file are used to generate a population from scratch.

3.1 Challenges

The population generator was challenging in an unusual way, compared to the rest of the project: integrating it into the other pieces was simple, but

the requirements were trickier to adhere to.

We initially wrote the population generation code following the first version of the specification as closely as possible, but we (and other groups) concluded that it was difficult, if not impossible, to generate populations for which all the requirements it specified held simultaneously. This part of the spec ended up getting rewritten, meaning we had to start over from scratch.

Generating towns that lie geographically between the specified cities was harder than expected. We measure the convex hull spanned by the pre-existing cities, and sample random points inside it. This way, the simulation area for a geo-distribution profile like `belgium_population_major.csv` is roughly Belgium-shaped, but this only works because Belgium is approximately convex to begin with.

3.2 Implementation details

Our generator returns a twofold result: the generated `Population` contains a collection of person data fitting the parameters defined in the model, for the simulator, and also an `Atlas` object, which contains geographical data about the generated towns and cities, for the Visualization tool to use.

Lynn wrote both the initial and final versions of the generator.

4 Multi-region simulations

We implemented single-process, shared-memory multi-region simulations, but didn't have for multi-process/multi-machine simulations.

4.1 Single-process, shared-memory multi-region simulations

The process of implementing single-process, shared-memory multi-region simulations consisted of two parts: refactoring the codebase in preparation for multi-region simulations and actually implementing multi-region simulations. The former task proved to be especially challenging as Stride simulations used to rely on (race-conflict–prone) global variables and adding

or removing people from a simulation was only supported at simulation set-up time.

Once refactored, the codebase was updated with support for multi-region simulations. This consisted mostly of designing data structures and adding logic to exchange people between simulations.

The user manual gives a good overview of what multi-region simulations can do.

Jonathan implemented multi-region simulations.

4.2 Notes on MPI-based multi-region simulation

We had planned to implement MPI-based multi-region simulation setup and transfer by serializing the state of a single-region simulation as a checkpoint, sending the checkpoint to another machine or process as a byte stream and then deserializing the checkpoint into a simulation. We imagined that using HDF5-based checkpoints over an ad-hoc format would save us work, reduce code duplication and reduce serialized checkpoint sizes.

Sadly, checkpointing support materialized later than expected—key single-region simulation checkpointing functionality landed in the master branch just after the beta presentation. Even then, the checkpointing API was not yet entirely suitable for usage in MPI-based multi-region simulations.

Our struggles with integrating checkpointing in MPI-based multi-region simulations can be attributed in part to the fact that restoring a checkpoint does a whole lot more than just recreating the state of a single-region simulation. Furthermore, some of our assumptions about HDF5's capabilities were invalidated. For example, we took for granted that HDF5 would be able to produce and consume in-memory byte streams. This turned out not to be the case, further adding to the list of workarounds we would need to implement to deliver a *prerequisite* to MPI-based multi-region simulation support.

Faced with daunting technical difficulties and a tight schedule, we decided to forgo MPI-based multi-region simulations and focus our efforts elsewhere.

5 Visualization tool

We went through various different approaches to implementing visualization during the course of our project.

5.1 Component-based approach

Our first approach was to simply extend the simulator with a component that would allow it to render directly to a window as it ran. We quickly decided on *Awesomium* as a suitable visualization library. *Awesomium* boasted to be a HTML-based visualization component compatible with both Windows and Linux builds. We made a very small prototype to verify that the library worked, but when we eventually tried to use it to create a visualization window we discovered that its functionality on Linux had been largely overstated. We found it to be completely useless for our purposes, so we removed it from the project.

We tried to find a replacement in *Qt*, which also boasts an option for HTML-based visualization and multi-platform support. However, after many attempts over a few days, we found it to be very difficult to successfully integrate *Qt* in our project.

These setbacks ended up giving us more room to think about our approach, and instead of moving on to a third visualization library we began to consider the potential downsides of this approach entirely: This approach would not easily allow us to locally visualize a simulation running on a remote server, and it would also make it difficult to implement a way to start the visualization in the middle of a simulation.

5.2 Server-based approach

With these issues in mind, we considered a different approach: Extending the simulation with an HTTP server component that would act as an interface allowing external applications to request information. Concretely, the external application would be a web page running in a browser.

This approach would resolve the issues with the component-based approach: Visualizing a simulation running on a remote server would only require setting up a VPN connection. As for starting a visualization in the

middle of the simulation's run, the fact the simulation and visualization are independent applications automatically requires and also simplifies this feature.

To this end, we started with integrating *POCO* into the project, a library allowing us to create the desired REST API which would be the interface between the simulation and the visualization.

Issues with this approach, however, was that it did not allow a practical way to visualize a simulation after it had closed, and that the demand for a REST API added unnecessary complexity to implementing the communication between the simulation and the visualization.

Due to these factors we decided to remove *POCO* from the project, and led us to our final approach to visualization.

5.3 File-based approach

The file-based approach means the visualizer simply outputs all relevant visualization information as a file so that an external tool can visualize it. Concretely, it outputs the data as a *json* file, and the visualization tool is a web page that runs in a browser.

This approach improves upon the previous ones as now the visualization data exists in a more permanent medium. The simulator is no longer required to be running in order to get access the visualization. Visualization of a remote simulation is still possible, although it now requires synchronizing the output file locally.

A remaining hurdle was the desire to get visualization while the simulation is still running, as initially the file was only written at the end of the simulation, which could potentially take a while. The solution to this was to make the simulator periodically append new data to the output file, and giving the visualizer the ability to detect and read these changes in order to keep its view up-to-date.

A further advantage of this approach is that the interface between the simulation and the visualization is a simple *json* format, described in the user manual. This allows for easy adjustments to the file format that don't require complex changes in the implementations of the visualizer or the simulator.

Sibert implemented visualization.

6 Overarching tasks

6.1 Workflow

Our workflow was heavily reliant on the tools offered by GitHub for team collaboration.

Each team member contributed to the project through feature branches on their own fork. Whenever a feature was finished, a pull request was made to the `flu-plus-plus` repository. Such a PR could only be merged into the master branch after approval by both our CI server (see next section) and at least one other team member (using GitHub's code review interface).

This approach forced us to keep track of each other's work, which is invaluable in a highly interdependent project with as many interacting functionalities as Stride.

6.2 Tests and CI

We initially set up both Jenkins and Travis CI for our project, but later abandoned Jenkins in favor of an all-Travis workflow.

Whenever someone pushes at least one commit to their fork of the master repository, Travis clones the appropriate branch of said fork. It subsequently follows the instructions in our `.travis.yml` file, which can be summarized as the following steps:

1. Download or build external dependencies such as CMake, a C++ compiler, Boost, ...
2. Build Stride
3. Run all tests
4. Generate a comparison report and push it to `https://flu-plus-plus.github.io/gtest-reports/comparison.html`.

The above happens on every push for the following targets:

Operating system	**Compiler**	**Parallelization library**
Ubuntu Trusty Tahr	GCC 6.0	OpenMP
Ubuntu Trusty Tahr	Clang 4.0	TBB
Ubuntu Trusty Tahr	Clang 4.0	STL
Mac OS X Sierra	AppleClang (XCode 8.3)	STL

Travis CI gives us a blank virtual machine on every build, except for some cached folders. Explicitly downloading or building dependencies ensures that our forks of Stride don't bit-rot away by depending on the specifics of the machine hosting a Jenkins build. Cached folders allow us to cache, e.g., compiled Boost binaries, which saves us time on the next build and reduces the strain we put on Travis CI's resources.

In addition to building our commits, Travis also makes sure that our pull requests can be merged into the master repository without regressing on any of the tests. If `git` can merge a pull request's contents merged into the master repository, Travis will clone the master repository, merge in the pull request locally, and run the tests, as per usual.

All of this is integrated nicely in GitHub's pull request workflow: a status badge tells us if a pull request builds and passes all tests. Additionally, we configured GitHub's *protected branches* feature to require that the Travis build always succeed before merging a pull request into the master branch of our master repository. Barring nondeterminism, this configuration ensures that the master branch's tests never fail.

6.3 Merging with the upstream Stride repository

One unique hurdle we faced was the assignment of merging our changes to the code with those made upstream by the members of the Stride team during the project. We knew the two codebases would keep diverging anyway, so we held this task off as far as we could. Throughout the majority of the project, however, this demotivated us from refactoring or formatting any of the source files provided to us: we knew doing so would just cause us more work during the final upstream merge. This meant we had to work with an unformatted and sometimes ill-documented codebase as our foundation, without getting an opportunity to fix anything until the very end.

In the end, the merge itself was performed by looking at a `diff` between

a commit from around the start of the project and the latest `comp/master` commit, and replaying those changes onto our code manually. The code we had to merge with was largely undocumented and lacked tests, so it was often difficult to carry over a change without knowing the intent behind it (or being able to verify that we merged it in successfully.)