Case- and Performance Study with the Extended Stride Simulator.

Episim

June 2018

Abstract

In this paper we will discuss the influence of our extensions to the stride simulator. Specifically we will analyse the performance w.r.t. run-time and memory-usage as well as investigate the impact of municipalities, i.e. the impact on the simulation's results when we would defragment some fraction of the cities. Before we start, we'll summarize the additions we've made to stride and briefly explain the function of the most important components.

1 Extensions to the stride simulator

The past few months we have made some additions to the stride simulator which we will briefly describe in this part of our paper.

1.1 Models

City

The City class represents an actual city in the simulator and essentially consists of a collection of communities and households along with information regarding all incoming and outgoing commute traffic. Some other data-members were introduced to reduce the amount of time needed for some operations, i.e. we're caching some information for faster retrieval afterwards.

Community

A Community has a type that can be either a school, college, workplace, primary or secondary community. It also consists of a collection of pointers to contact pools and a pointer to the city it belongs to. Since contact pools are the core concept of the simulator, this specialized container makes the management of these contact pools a bit easier.

Household

The Household class is another specialized container for a single contact pool. Aside from a pointer to its contact pool it also has a pointer to the city in which this household is located.

GeoGrid

The GeoGrid represents a geographical location for the simulator. In a nutshell, this class contains information such as its cities, models for the distributions to be used, the actual population and some cached items for faster retrieval.

1.2 Generators

In this section we will briefly give a description of how our generators work. The first generator is the GeoGridGenerator which, as the name suggests, generates a GeoGrid object according to the given configuration. The second generator is the PopulationGenerator which takes a GeoGrid object and generates a population for that particular GeoGrid object that was given.

Geogrid generator

A grid is generated either according to specifications found in a configuration file[1] or by passing a property tree. If the specified file cannot be found the default one (i.e. run_default.xml) is used instead. The actual generation is a two step process that goes as follows.
In the first step we initialize the data members, i.e. we read the cities, household model and commuting model from files mentioned in the configuration file. Once the entire configuration/initialization is done we move up to the second step. This step consists of generating the colleges, workplaces, schools and communities, all w.r.t the implied distributions from the configuration.

Population generator

Once the GeoGrid has been constructed, we can pass this to the PopulationGenerator. It will then generate a population according to the given GeoGrid's model.
In short, the PopulationGenerator will select a random city (according to the discrete distribution derived from population models of the cities) for which a household will be made. Next it selects a random household size according to the distribution inferred from the model. Finally it will choose a random household of the given size from the model and proceed with the actual generation of people for that household.

1.3 Utils

As we progressed with the project we noticed that we had several functionalities that were needed but did not really belong to any of the models. In this section we wil give a brief summary of them.

Parser

Parses information about cities, commutes and households from a given file.

GeoGridFileWriter

Writes a given geogrid to the following files, usually located in the `<output_prefix>/GeoGrid` directory unless you use the seperate writer-functions in which case you have to specifiy a path or a file-name:

- cities.csv (current cities of the grid)
- commuting.csv (current commuting model of the grid)
- households.xml (current households model of the grid)
- population.csv (current population of the grid)
- communities.csv (current communities of the grid)
- RNG-state.xml (current state of the grid's RNG)
- geogen.xml (current configuration for geogen)
- config.xml (current configuration of the simulator)

[1] The first argument in Generate(boost::filesystem::path& config, const bool contactFile = false)

Utils

Contains functionality that allows us to

- iterate over the 'Fractions'[2] enum fractions (either all fractions or only age fractions);
- iterate over the 'Sizes' enum;
- check wheter a file exists;
- generate N random numbers from a given distribution vector and random number engine;
- generate N random numbers between two values given a certain random number engine;
- check wheter a given distribution contains only zeros and transform it to a uniform distibution if so;
- return the fraction category for a certain age;
- write a given pTree to an xml file;
- use the constant ϵ.

Test summarizer

Contains the functionality that generates a summary of all tests located on a given path.

1.4 GUI

As requested, a graphical user interface was made. It can be activated by invoking stride with the `-e simgui` option. We used the Qt SDK to write our GUI. The following functionallity is provided:

- Running the population generator. Generates a GeoGrid and population.
- An interactive world map. The cities in the GeoGrid are represented as circles on a world map. on mousehover over a circle, a popup window appears with some general information about the city.
- Clickable cities. Clicking on a city circle will provide detailed information about that city in the left toolbar.
- Multi selectable cities. By holding shift and dragging the mouse we can select/unselect multiple cities at once.
- Running the simulator. Run the stride simulator. We can choose the stepcount of the simulator, or run all steps provided in the configuration.
- 2 Configuration tabs. Here we can specify all settings of both the population generator and the stride simulator.

1.5 Chart

By using QtChart, the important characteristics of the generated population can be visualized graphically. Bardiagram is used to visualize the age distribution, households' distribution, workplaces' distribution and by using pie-chart the population density of the generated population is visualized.

[2]SCHOOLED (aged 3-17 years), ACTIVE working population (aged 18-64 years), YOUNG people (aged 18-25 years), MIDDLE_AGED (aged 26-64 years), TODDLERS (aged 0-2 years), seniors (OLDIES, aged 65+ years), STUDENTS (aged 18-25 years), COMMUTING_STUDENTS (same ages as STUDENTS), COMMUTING_WORKERS (same ages as ACTIVE)

2 Performance analysis (run-time)

2.1 Setup

In our performance analysis we will investigate the influence of 3 parameters:

- Population size
- Number of threads
- Random engine

For the first parameter we expect slightly more than a linear increase in run-time. The reasoning behind this is simple. Since we're only generating n people, the time complexity is clearly $O(n)$. Though we're not keeping in mind that we sometimes need to search for communities and thus the time-complexity increases. However if we look closely to `PopulationGenerator::GetRandomCommunities` we see that radius is an unsigned int (32 bits). Mind that the radius is shifted and this can only happen 32 times at most, which means we have an upper bound. Another function that contributes to this extra time-complexity is `PopulationGenerator::GetNearestColleges`. Here we see a for-loop that is bounded by the number of cities m that have colleges. This means that in the worst case scenario we'd have $O(nm)$ time-complexity, but since m is usually very small compared to n and the fact that this function can only be called if we're dealing with a student makes this somewhat neglectable. For more details on the implementation, simply refer to the PopulationGenerator.cpp which can be found in our github repository under main/cpp/popgen-Episim/generators.

The second parameter is hard to estimate its impact but obviously we expect a faster run-time when more threads are enabled. For the last parameter we expect no changes in run-time which seems pretty straight forward, however we will take the time to actually verify that this is in fact the case.

In order to check the effects on run-time we decided to use `gprof`. We used this program to generate a flat profile and a call graph. A flat profile indicates how much times is spent on each function and how many times it was called. The call graph gives a view of how many functions a function calls itself. To use `gprof`, we first ran our executable (i.e. `./bin/stride -e geopop`. After this we used `gprof -b -p ./bin/stride` to generate the flat profiles and `gprof -b -q ./bin/stride` to generate the call graphs. The `-b` flag used in both commands is only used to improve readability and does not changes the data. All profiling data regarding run-times was collected on a single machine, avoiding inconsistencies in the measurements. Mind that `./bin/stride -e geopop` uses the default configuration (`run_default.xml`) which is also provided in the zip-file under the folder `configs`.

2.2 Varying the population sizes

In this part we will discuss the influence of the population size on the performance of the simulator. The population sizes we used go from 1,000,000 to 45,000,000 in steps of 4,000,000. For each step we ran the simulator 10 times in order to be able to compute averages which we'll use for our analysis. Furthermore we'll discuss the flat profiles generated by `gprof` which can also be found in the zip under the folder `gprof_output/pops/*/FlatProfiles`.

Configuration

To investigate the impact of the population size on the run-time, we kept the configuration the same throughout all simulations except for the number of people to be generated for the GeoGrid. To do this we had to make serveral geopop configurations with different population sizes. These have been included in the zip-file under the folder `configs`. The names of these configuration files were then used to override the `geopop_file` attribute through the command-line interface, i.e. using the `-o` option. This removes the need to make seperate stride configurations.

Run-time

When we examine the simulator's output (found in the folder `gprof_output/pops/*/StrideOutput` within the zip-file) we can see the time that was needed for the simulator to generate the population as well as the time spent during the actual simulation. Computing the average of the 10 runs for each step yields the following table:

#people to be generated in million	Time needed for popgen in seconds	Time needed for simrunner in seconds
1	3.750834	6.990238
5	20.416040	36.799291
9	37.870900	69.761764
13	55.834000	102.869547
17	74.949770	137.022907
21	92.442740	169.782989
25	110.163200	204.340417
29	129.173900	243.729355
33	147.982100	282.761058
37	166.931100	319.494965
41	184.736000	364.223016
45	205.881300	401.548743

Table 1: Average run-times for population sizes

At first sight it looks like our expectations are being met, i.e. we can see a linear increase in run-time. We can also confirm this by looking at Figures 1 & 2 on the next page. This is both true for the run-time of population generator as well as the run-time of the simulation itself. Obviously the difference between the two run-times is irrelevant since the run-time of the simulation itself is also affected by the number of days to be simulated.

However a closer look at the average run-times reveals that we there is in fact slightly more than just a linear increase, again as we expected. This can be verified by dividing the number of people to be generated by the average run-time. This yields the following table:

#people to be generated in million	Time needed for popgen in seconds/million	Time needed for simrunner in seconds/million
1	3.750834	6.990238
5	4.083208	7.359858
9	4.207878	7.751307
13	4.294923	7.913042
17	4.408810	8.060171
21	4.402035	8.084904
25	4.406528	8.173617
29	4.454272	8.404461
33	4.484306	8.568517
37	4.511651	8.634999
41	4.505756	8.883488
45	4.575140	8.923305

Table 2: Average run-times per 1 million people

Clearly the average time to generate 1 million people increases along with the number of people to be generated. The same applies for the actual simulation, though this value has little meaning which is why we won't discuss it. Looking back to the average run-time for generating 1 million people, we can notice something strange for populations of sizes 17, 21 & 25 million. The averages stay roughly the same for those sizes, however with the current data available we can't make any conclusions as to why this phenomenon presents itself. We can only assume this is the result of an unlucky combination of parameters in the configuration, though more scenarios should be investigated to verify this assumption. At last, we will derive a function which approximates the run-time for both the population generator as well as the actual simulation. We'll start off with the population generator, followed by the actual simulation and then proceed to combine these two to get a rough estimate for the total run-time.

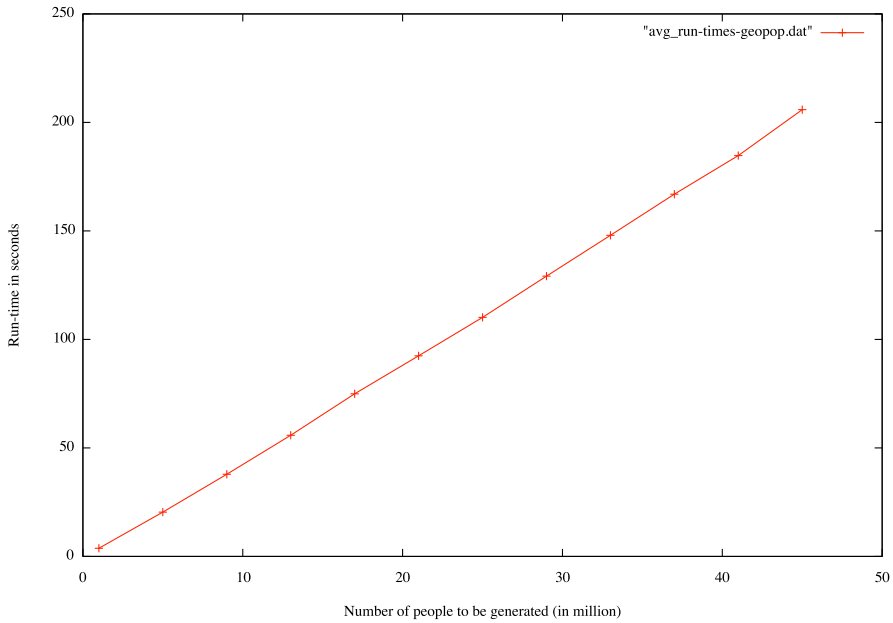

Figure 1: Run-times for population generator

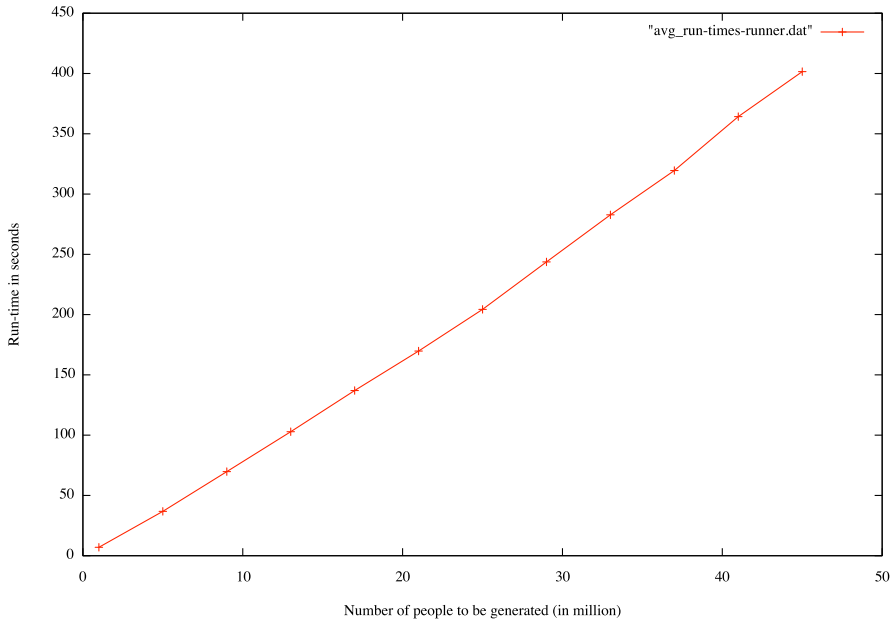

Figure 2: Run-times for the actual simulation

Using linear regression on the average run-times, we get the following estimates:

$$\begin{aligned} RuntimePopgen &= -2.977 + 4.586 \cdot \#Pop \\ RuntimeRunner &= -11.920 + 8.994 \cdot \#Pop \\ RuntimePopgen + RuntimeRunner &= -14.897 + 13.58 \cdot \#Pop \\ &\approx TotalRuntime \end{aligned}$$

Where $\#Pop$ is the number of people to be generated in million. Now mind that this is only valid for the machine on which we produced these results. In order to generalize the estimate, we would need a modifier x which indicates how much faster or slower a particular system would be w.r.t. our machine. A value $x > 1$ would indicate a the machine is slower than ours while $0 < x < 1$ would represent a faster one. Subsequently an identical machine would have $x = 1$. To sum everything up, we'll show all

Calls made

`stride::Person::Update` and `stride::Health::Update` each account for 26.3% of all calls when the population becomes larger then 1,000,000[3]. This means that these 2 functions together account for more than half of all the calls. Other functions with notable contributions are: `std::Sp_counted_base<(__gnu_cxx::_Lockpolicy)2>::_M_release()`[4], `stride::Infector<(stride::ContactLogMode::Id)1, ...>)`[5] and `stride::ContactPool::SortMembers()`[6]. The function `stride::ContactPool::AddMember(stride::Person* const)` stays stable at ≈1.9% for all population sizes.
A notable change occurs for `std::function<trng::uniform_int_dist::result_type()> stride::util::RNManager::GetGenerator<trng::uniform_int_dist>(trng::uniform_int_dist const&)`. This function sees an increase in number of calls from 1.04% to 3.07%.
The charts from which the percentages are from can be found in the zip under the folder `diagrams/pop/calls`.

Time spent in seconds on each function call, descendants excluded

We see that `stride::Immunizer::Random(stride::util::Segmented_Vector ...)` stays a dominant function with a contribution of 52.2% for 1,000,000 to 76.9% for 45,000,000[7]. Further we see that `stride::PopulationGenerator::ClassifyNeighbours()`[8] stays constant at 0.02 seconds and `stride::Population::GetInfectedCount()` sees an increase from 0 to 0.18 seconds. Apart from this no notable changes occur.
The charts from which the percentages are from can be found in the zip under the folder `diagrams/pop/self s_call`.

Time spent in seconds on each function call, descendants included

If we include the time spent in each descendant we see that there are 4 functions that take exponentially more time with larger populations. These functions are `stride::SimRunner::SimRunner(...)`[9], `stride::Sim::Create(...)`[10], `stride::Sim::Sim()`[11] and `std::_Sp_counted_ptr_inplace<stride::Sim::Create(...`[12].All other functions see a small increase.
The charts from which the percentages are from can be found in the zip under the folder `diagrams/pop/total s_call`.

Time accounted for by a function alone

The first thing that can be noticed is that `stride::ContactPool::SortMembers()` is by far the function with the longest running time going from 2.17 seconds for 1,000,000 people to 175.17 with 45,000,000 people[13]. Another function that undergoes noticeable changes is `stride::PopulationGenerator::GetRandomContactPool(std::vector<stride::Community, std::allocator<stride::Community> > const&)`: where this function itself only took 0.2 seconds when the population only consisted of 1,000,000 people it increases to 26.88 seconds for 45,000,000. Apart from these two functions no noticeable changes can be distinguished[14].
The charts from which the percentages are from can be found in the zip under the folder `diagrams/pop/self seconds`.

[3] When the population is 1,000,000 each of them makes up 27.1% of all calls
[4] 12.5% at 1,000,000 and 12.2% in all other cases
[5] 12.4% at 1,000,000 and 12% in all other cases
[6] 12.4% at 1,000,000 and 12% in all other cases
[7] Or 0.12 seconds to 9.98 seconds
[8] A function that has the second highest contribution for 1,000,000
[9] 3.31 seconds to 229.02 seconds
[10] 3.3 seconds to 228.33 seconds
[11] 2.51 seconds to 174.23 seonds
[12] 0.62 seconds to 43.56 seconds
[13] But if we look at it procentualy there is only a small increase from 42.8% to 52.2%
[14] All functions see a small increase in their time, but nothing that can't be expected

2.3 Varying the number of threads

In this part we will discuss the influence of the number of threads used on the run-time of the simulator. We will discuss the impact of 1 thread, 2 threads and 4 threads. For each step we ran the simulator 10 times in order to be able to compute averages which we'll use for our analysis. Furthermore we'll discuss the flat profiles generated by `gprof` which can also be found in the zip under the folder `gprof_output/threads/*/FlatProfiles`.

Configuration

To investigate the impact of the number of threads on the run-time, we kept the configuration (i.e. run_default.xml which can be found in the `config` folder within the zip-file) the same throughout all simulations except for the number of threads to be used. This number can be easily changed by using the `-o num_threads=..` flag when running the simulator.

Run-time

Following the same strategy as before, we can again compute the average run-times which gives us the following table:

#threads to be used	**Time needed for popgen in seconds**	**Time needed for simrunner in seconds**
1	17.669710	31.754954
2	13.739510	39.034853
4	9.669450	29.686544

Table 3: Average run-times for #threads

For the population generator we can see a decrease in run-time of $\approx 22\%$ for 2 threads and $\approx 45\%$ for 4 threads, which corresponds with what we expected, i.e. a decrease in run-time. The time needed for the actual simulation is a different story. Here we see the run-time staying roughly the same for 1 & 4 threads, while for 2 threads we surprisingly have an increase of $\approx 23\%$ in run-time. We're assuming this increase is caused by a problem with the machine (Linux Ubuntu 18 with GCC7) on which these results were produced. The reason for this assumption is the fact that we quickly the same scenario on MacOSX (10.13.3) using Clang7 which resulted in run-times that were lower for 2 threads, which in turn resulted into little differences in run-time among the 1, 2 & 4 threads. Testing this scenario on a third machine again with Linux Ubuntu 18 & GCC7 gave us a decrease in time, thus we got 3 different results which makes it difficult to give an estimate for the run-time of the actual simulation. The reason as to why we're getting different results on all 3 machines is unknown to us.

However that doesn't keep us from estimating the run-time for the population generator. To do this we assume the relative decrease in run-time doesn't change significantly, no matter what scenario we're running. Then we take (100-22)% & (100-45)% which gives us 78% & 55%. Expanding the estimate of the run-time for the population generator that we derived earlier, we get the following:

#threads	**Estimate to be used for popgen**
2	$0.78 \cdot RuntimePopgen = 0.78 \cdot (-2.977 + 4.586 \cdot \#Pop)$ $\Leftrightarrow 0.78 \cdot RuntimePopgen = -2.322 + 3.577 \cdot \#Pop$
4	$0.55 \cdot RuntimePopgen = 0.55 \cdot (-2.977 + 4.586 \cdot \#Pop)$ $\Leftrightarrow 0.55 \cdot RuntimePopgen = -1.637 + 2.522 \cdot \#Pop$

Table 4: Estimated run-times for #threads

Time spent on each function call, descendants excluded

`stride::Immunizer::Random(stride::util::SegmentedVector<stride::ContactPool` sees a slight increase from 0.63 seconds to 0.68 seconds, even though this means a decrease from 70.8% to 45.6%. This can be explained by the increase of time spend on `stride::PopulationGenerator::GenerateHousehold(unsigned int)` from 0 seconds to 0.38 seconds or 25.5%.
The charts from which the percentages are from can be found in the zip under the folder `diagrams/threads/self s_call`.

Time spent on each function call, descendants included

No matter how many threads we use, `SimRunner::SimRunner(boost::property_tree ...)` is always the function with the longest total active time if we include time spend in descendant calls[15]. Other functions with notable contributions are `Sim::Create(boost::property_tree ...)` and `Sim::Sim()`[16].
The charts from which the percentages are from can be found in the zip under the folder `diagrams/threads/total s_call`.

Time accounted for by a function alone

As is to be expected when increasing the number of threads to be used we see that `stride::ContactPool::SortMembers()` takes less time[17]. Of course this decrease[18] comes at a certain cost: we see an increase in the time send on `std::_Sp_counted_base<(__gnu_cxx::_Lock_policy)2>::_M_release()`[19]. As this increase is of the same size as the decrease of `stride::ContactPool::SortMembers()` all other decreases in time are pure profit.
The charts from which the percentages are from can be found in the zip under the folder `diagrams/threads/self seconds`.

Conclusion

If further gains in run-time are desired we suggest looking at the mentioned functions to make further improvements.

2.4 Varying the random engine

In this part we will discuss the influence of the random engine on the performance of the simulator. We used the LCG64, LCG64_SHIFT, MRG2, MRG3, YANR2 and YARN3 engines. For each step we ran the simulator 10 times in order to be able to compute averages which we'll use for our analysis. Furthermore we'll discuss the flat profiles generated by `gprof` which can also be found in the zip under the folder `gprof_output/engines/*/FlatProfiles`.

Configuration

To investigate the impact of the random engine on the run-time, we kept the configuration the same throughout all simulations except for the random engine to be used. The engine can be easily changed by using the `-o rng_type==..` flag when running the simulator.

Run-time

Looking again at the simulator's output which we can find in the `output_gprof/engines/*/StrideOutput` folder which is provided in the zip-file, we compute the average run-times once again. The following table shows that the run-times are more or less the same, however a small difference is noticable for YARN engines. Assuming this is not a coincidence, we can conclude that the YARN engines have slightly worse performance compared to the rest.

[15] 29.9% for 1 thread, 24.3% for 2 threads and 24.2% for 4 threads
[16] Respectively 29.7% and 22.7%, 24.3% and 22.7%, 24.1% and 18.3% for 1 thread, 2 threads and 4 threads
[17] A reduction from 10.61 seconds or 43.8% to 7.75 seconds or 31.1%
[18] Along with all other decreases
[19] from 0.78 seconds to 3.53

RNG type to be used	Time needed for popgen in seconds	Time needed for simrunner in seconds
MRG2	17.235320	31.061595
MRG3	17.441370	31.239935
LCG64	17.729870	31.705904
YARN2	18.287590	32.146674
YARN3	18.187120	32.156198
LCG64 SHIFT	17.180660	31.171931

Table 5: Average run-times for RNG types

Calls made

When we look at the number of calls made when using each of the different engines we see that there is no noticeable difference between LCG, LCG_SHIFT, MRG2 and MRG3 procentually[20]. The 2 YARN engines both see a large increase in calls to `std::Function_Handler(int ...)`[21] and `std::Function_Handler(double ...)`[22].
The charts from which the percentages are from can be found in the zip under the folder `diagrams/engines/calls`.

Time spent on each function call, descendants excluded

There are three functions with a dominant contribution: `stride::Immunizer::Random(...)`, `std::_Sp_counted_ptr_inplace<stride::Sim::Create(boost ...)>` and `stride::Population::GetInfectedCount() const`.

Engine	`Immunizer::Random(...)` in %	`GetInfectedCount()` in %	`_Sp_counted_ptr_inplace<...>` in %
LCG	71.9	24.7	2.25
LCG_SHIFT	71.6	25	2.27
MRG2	71.1	25.6	2.22
MRG3	70.8	25.8	2.25
YARN2	72.8	23.9	2.17
YARN3	72.6	24.2	2.11

It is clear by looking at this table that once again the YARN engines are not preferable. It is also clear that the MRG engines are slightly better than the LCG engines. Note that we're only interested in `Immunizer::Random(...)` since the other two functions don't use a random number generator.
The charts from which the percentages are from can be found in the zip under the folder `diagrams/engines/self s_call`.

Time spend on each function call, descendants included

There are two functions with a dominant contribution: `stride::Immunizer::Random(...)` and `std::_Sp_counted_ptr_inplace<stride::Sim::Create(boost ...)>`.

Engine	`Immunizer::Random(...)` in %	`_Sp_counted_ptr_inplace<...>` in %
LCG	17.6	81.6
LCG_SHIFT	17.6	81.5
MRG2	18.4	80.8
MRG3	18.1	81.1
YARN2	19.2	80.1
YARN3	19.8	79.4

[20] Though the actual numbers may vary slightly
[21] From 1.02% to 4.08%
[22] From almost negligible to 5.24%

Once again the YARN engines are to be avoided.
The charts from which the percentages are from can be found in the zip under the folder `diagrams/engines/total s_call`.

Conclusion

The YARN engines should not be used if performance has to be at its best. Instead we suggest the use of one of the MRG or LCG engines as they have slightly better performance then the YARN engines.

2.5 GeoGrid's generation

We are also interested in the runtime performance of the generation of GeoGrid, though we expect this impact to be minimal. We will be generating and assigning schools, colleges, workplaces, primary- and secondary communities to different locations. The number of these will depend on the total population and the fractions of the population which are to be assigned to these places. It looks like the time complexity is linear for geogrid. As we are using a random number generator with some discrete distribution to assign schools, colleges, etc. to the different locations, the choice of the random engine can also have impact on the run time. For this analysis we used benchmarking framework `Myhayai`. This benchmarking is done in MacOSx (10.13.4) using clang-902.0.39.2 compiler.

To see the influence of the population size we varied the population from 1 million to 45 million in steps of 4 million. All the other factors like number of threads, the random engine used, other information that is read from the configuration files are kept constant. Every population count is run ten times and median is taken into account for the analysis.

#people in million	**Time needed for GeoGrid in seconds**
1	0.403
5	0.549
9	0.999
13	1.022
17	1.033
21	1.054
25	1.510
29	1.663
33	1.676
37	1.916
41	1.967
45	2.132

Table 6: Average run-times for GeoGrid with variable population sizes

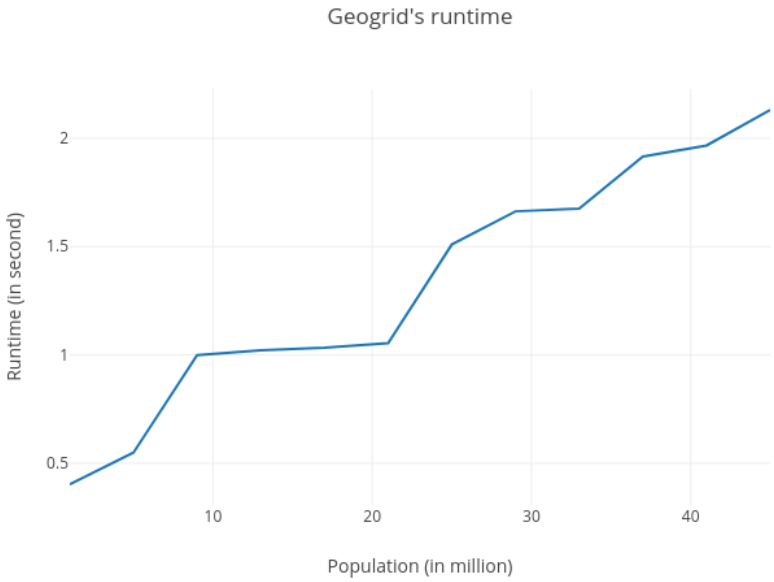

Figure 3: Runtime for GeoGrid's generation

As expected, the time needed to generate a GeoGrid is almost neglectable compared to the population generator. Obviously the run-time increases with increasing population because we will need more communities to assign the people and therefore there will also be more contactpools. But if we look carefully it shows, we will see the runtime per million population decreases with an increasing population which will be made clear by the following table:

#people in million	**Runtime per million in seconds**
1	0.403
5	0.1098
9	0.111
13	0.0786
17	0.0608
21	0.0501
25	0.0604
29	0.057
33	0.0508
37	0.0517
41	0.0480
45	0.04737

Table 7: Average run-times for GeoGrid per million people

We can see from table 7 that the runtime required per million people decreases as the number of people increases. The reason behind that is there are certain constant factors which influence the runtime while generating GeoGrid (the setup time), for example the time required to read the data from files or time required to calculate distances. For the small population the most of the runtime is occupied by the setup time.

By using linear regression on the average runtimes of GeoGrid's generation, we will get the following estimates:

$$RuntimeGeogrid = 0.4461 + 0.03830 \cdot \#Pop$$

The first constant value (0.4461) in the formula is an approximation for the run-time needed by the setup. The second one (0.03830) represents the approximated run-time needed to generate schools, communities, etc. for a population of 1 million.

We would also like to analyze the influence of random engines to the geogrid. For that purpose we will vary random engines and everything else will remain the same. The generation is run ten times as well.

RNG type to be used	**Time needed for geogrid in seconds**
MRG2	0.500
MRG3	0.581
LCG64	0.556
YARN2	0.515
YARN3	0.534
LCG64 SHIFT	0.543

Table 8: Average run-times for RNG types

We can clearly see from table 8 that random engines do not have that much of impact on the runtime. We can conclude that for the generation of geogrid the random engine doesn't have influence on the performance, surprisingly also for the YARN engines.

2.6 Conclusion

At this moment the simulator has a decent runtime, but we are sure that further time gains can be made by further investigating `stride::ContactPool::SortMembers()`, `stride::PopulationGenerator::GetRandomContactPool(std::vector<stride::Community, std::allocator<stride::Community> > const&)`, `stride::Population::GetInfectedCount()`, `stride::ContactPool::AddMember(stride::Person* const)`, `stride::Immunizer::Random(...)`, `stride::PopulationGenerator::GenerateHousehold(unsigned int)`, `SimRunner::SimRunner(boost::property_tree ...)`, `Sim::Create(boost::property_tree ...)` and `Sim::Sim()`.
Further we can only advise to use one of the MRG or LCG engines as the YARN engines have a negative impact on the total runtime.

3 Performance analysis (memory-usage)

3.1 Setup

In this section we will analyze the space complexity of stride with our integrated parts. We will use the process image size as a metric of the space complexity of our application. We used PAPI[2] (Performance Application Programming Interface) as tool to find the process image size and will investigate the influence of the same 3 parameters as in the previous section:

- Population size
- Number of threads
- Random engine

Predicted outcome

Population: We expect a linear relation between the population size and the image size. This relation is expected to be visible no matter the value of the other two parameters. This is because the amount of data structures (schools, households, persons...) made in the population generator is the amount needed to fit the given population. So we expect $\mathcal{O}(n)$ space complexity with n the population and the two other parameters constant.

Number of threads: Since we implemented threading making use of Open MP constructs, it is expected that the compiler will generate extra code to handle things like threaded loops. This happens at compile time and not at run-time, so the amount of threads specified should not affect the process image code size. It could be possible that extra variables are needed for thread management and the process image size will increase if we work in a multi-threaded environment.

Random engine: The random engine used should have a minimal impact on the amount of memory needed. The size of the population will stay the same no matter what generator we will use. The total amount of data structures made (like schools, workplaces...) is based on the average size and population parameter. The random engine is not used to decide the amount and size of those structures.

3.2 Data

What follows is a summary of the collected data. Each table contains the gathered data for a thread. Population amount is found left, process image is in kilobyte.

One Thread

Population	lcg64	lcg64_shift	mrg2	mrg3	yarn2	yarn3
1 000 000	864812	864584	865356	865008	864956	865072
2 000 000	864816	864652	865368	865076	865024	865140
3 000 000	864820	864656	865368	865080	865028	865144
4 000 000	864828	864656	865376	865076	865028	865148
5 000 000	864832	864664	865384	865080	865024	865144
6 000 000	864840	864668	865388	865080	865028	865140
7 000 000	864848	864668	865392	865076	865024	865144
8 000 000	864852	864672	865396	865076	865024	865140
9 000 000	864860	864672	865404	865080	865024	865144
10 000 000	864864	864676	865416	865076	865028	865148

Two Threads

Population	lcg64	lcg64_shift	mrg2	mrg3	yarn2	yarn3
1 000 000	892372	888112	881704	875964	885868	885672
2 000 000	884320	887176	886616	884080	884836	886684
3 000 000	883940	886444	884372	882732	883796	887092
4 000 000	882920	887868	882656	886684	884760	884220
5 000 000	886556	884728	884632	887552	886080	881628
6 000 000	884036	819528	888860	885640	884092	885688
7 000 000	885432	819528	888860	882392	885168	881292
8 000 000	886332	819528	886556	884500	885700	887072
9 000 000	888976	819528	887556	882296	884832	882808
10 000 000	885200	819528	887228	885308	884976	880564

Three Threads

Population	lcg64	lcg64_shift	mrg2	mrg3	yarn2	yarn3
1 000 000	984480	982784	984756	984184	982908	979528
2 000 000	976696	988600	980644	985908	974988	990480
3 000 000	984616	989812	983928	989052	989136	984260
4 000 000	985796	988652	987088	986800	990688	975636
5 000 000	973824	985276	989060	986448	991484	986540
6 000 000	978564	989680	989608	989624	991496	987736
7 000 000	979184	983232	988384	987352	993232	977736
8 000 000	984916	988936	992420	992328	990120	984856
9 000 000	990180	989556	980168	992092	991156	984204
10 000 000	975896	987312	990224	995200	990084	982416

Four Threads

Population	lcg64	lcg64_shift	mrg2	mrg3	yarn2	yarn3
1 000 000	1004872	1007008	1002984	1006248	1006104	1009044
2 000 000	1003676	1007280	1003904	1005328	1006412	1005912
3 000 000	1007452	1004404	1006668	1007460	1008484	1004636
4 000 000	1008004	1004640	1004544	1007188	1007420	1005988
5 000 000	1006880	1003956	1006868	1011620	1004260	1006236
6 000 000	1005408	1008408	1004152	1003160	1002212	1001476
7 000 000	1004756	1006484	1008248	1003760	1006816	10039720
8 000 000	1005220	1006396	1004792	1007552	1004872	1006016
9 000 000	1005292	1006984	1002860	1005256	1006604	1001832
10 000 000	1005436	1002992	1008020	1003664	1004532	1002440

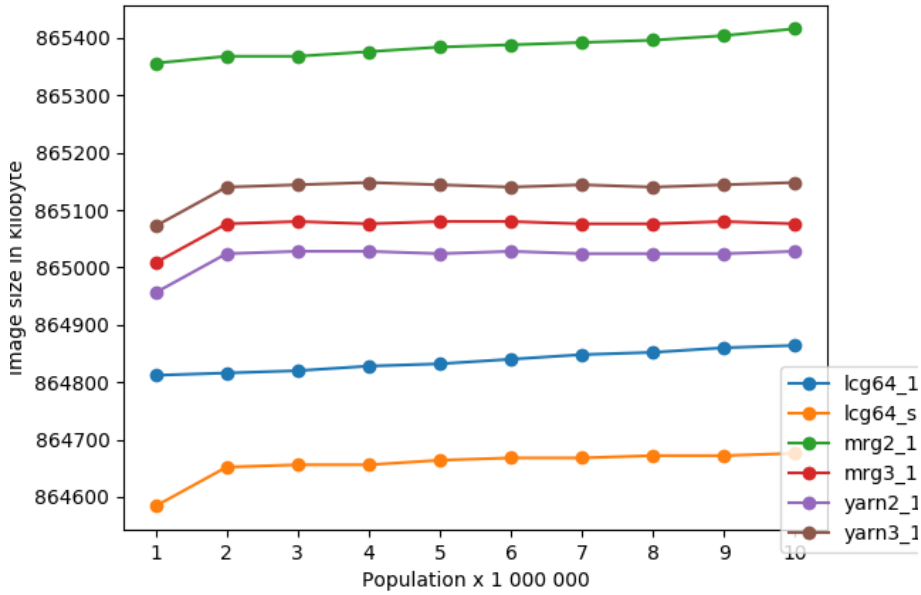

Figure 4: One thread memory usage

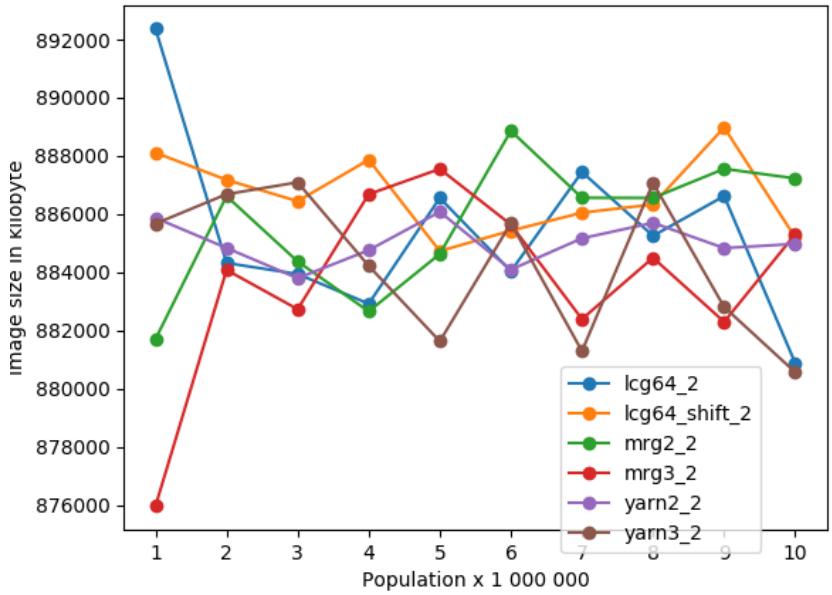

Figure 5: Two threads memory usage

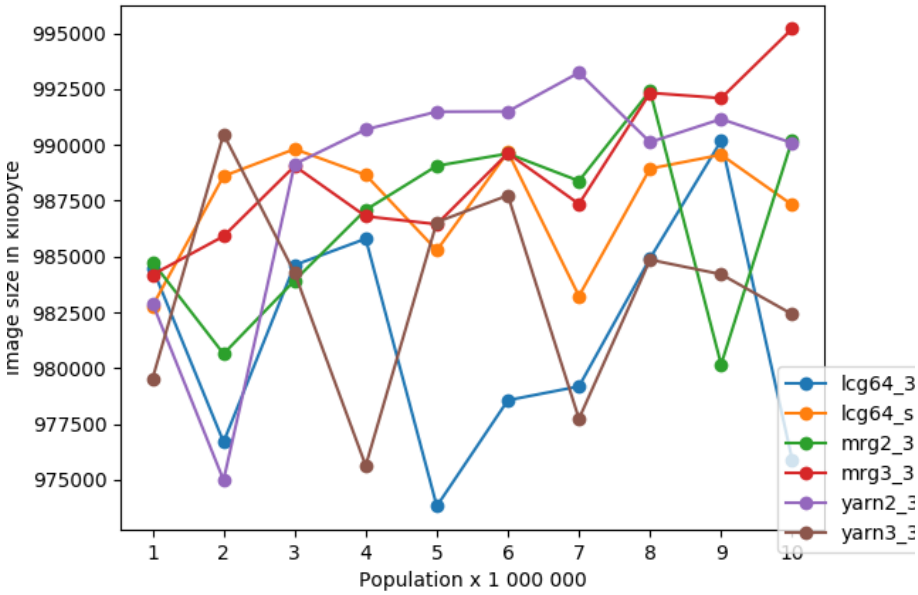

Figure 6: Three memory usage

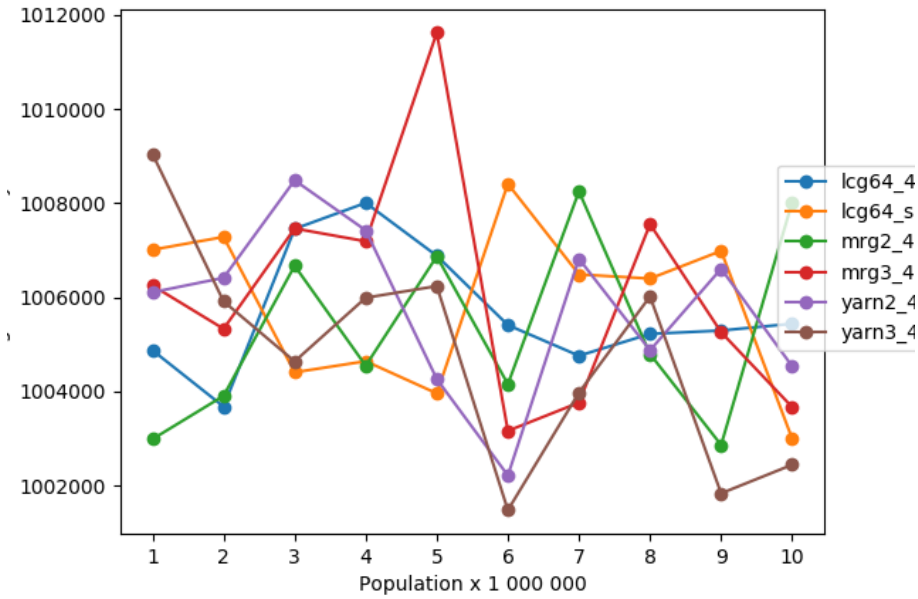

Figure 7: Four threads memory usage

3.3 Discussion

Population: We expected a linear relation between the population size and the process image. But our data does not represent this. Only the one thread data has some resemblance . With one thread: At some points we see a lower memory with a higher pop. but overall, there is a small memory increase. This is an indication that with bigger datasets we could have a more clear relation between population and memory. We also see a flex point (local maximum) at a population of 2 million. The moment we start to use multiple threads the process image size varies significantly with no clear pattern.

Number of threads: As expected, memory usage increases the more threads we use. We also observe that the moment we start to use more then one thread we don't see any relation between population count and process image size. Also there is no generator that uses more or less memory than another over the used sample size with more threads so no conclusions can be made.

Random number generator: Data shows a contradiction with our expectations. With one thread the process image size differs widely between the different random number generators. A possible explanation could be just variance between how the random number generators handle the default seed found in the config file. Another explanation would be that the RNG's themselves consume different amounts of memory. The third explanation could be that PAPI registered different memory usages on different times, since we always took samples for each RNG sequentially. The moment we start to use multiple threads there is nothing we can say about the "better" random number generators. Figures 5, 6 and 7 show this.

3.4 Conclusion

Population is probably linear related to memory usage. This can be seen if we use one thread. However, the used population data samples don't show significant differences.

For one thread, the data shows a relation between Process image size and random number generator uses. This is a contradiction with our prediction. Is this just the specific case with the initial seed? The moment the test is run on the reference machine? Or does mrg2 really need a lot more memory? Further experimentation is needed. With more threads there is no clear relation.

Threading adds a constant, as expected, to the process image size. This is good news in terms of scaling. The constants are small enough, this means we don't sacrifice to much memory if we want to speed up the simulation by adding more threads. Threading gave a unexpected variance in the process image sizes. Different (bigger?) datasets should be used to investigate this.

4 Case study

In this case study we will try to show there is a limited (if any) influence of the defragmentation of cities on the results of the simulator: if a city is split into smaller parts the commuters of the original city are divided equally over the new smaller parts. We don't expect that this will have a noticeable influence on the results.

For this study we will use a configuration which uses 1 thread and the MRG2 random engine and we will defragment cities if they contain less than 5% of the total population[23].

The following table gives an overview of the infected count on the simulation day if we use defragmentation or not.

Simulation day	No defragmentation	Defragment 2.5%	Defragment 5%
0	8683	8683	8683
5	9508	9514	9531
10	34802	34879	34549
15	58723	58900	58636
20	135245	135429	134102
25	240226	241111	239591
30	405513	407231	404606
35	573870	575775	574043
40	712256	713736	712168
45	788746	789357	789066
50	819307	819528	819658

We see that there is a small increase in results. This is however not due to defragmentation of the cities but caused by noise. When we defragment the cities we create a certain amount of new sub-cities. By doing so we not only change the amount of cities in our grid but by extension also the number of communities and contact pools. This may cause changes in the way the engine generates random numbers and by extension the rest of the simulator, thus giving an explanation for the noise we see here.

Another explanation may be that we now have more places from which commutes are possible to and from. This increase in possible origins and destinations may lead to an increased rate at which the population will be infected. An increased infection rate will lead to more infections and thus to higher infected counts if we increase the percentage of cities we want to defragment into sub-cities.

[23] The configuration files can be found in the zip under the folder `defrag-study/`

References

[1] Qt *https://www.qt.io/*

[2] PAPI *http://icl.cs.utk.edu/papi/index.html*