UPPSALA
UNIVERSITET

IT 12 002
Examensarbete 15 hp
Januari 2012

Parallelization and performance in simulation of disease spread by animal transfer

Fredrik Pasanen
Magnus Söderling

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Parallelization and performance in simulation of disease spread by animal transfer

Fredrik Pasanen, Magnus Söderling

E. Coli O157:H7 is a verotoxin producing bacteria commonly spread from cattle to human through the intake of contaminated food or water. In order to simulate the spread of the disease a Monte-Carlo simulation model were developed which did not have an optimal run-time were developed by another author. In order to decrease the run time the implementation were optimized and parallelized by using profilers in order to find the bottlenecks. This lead to an implementation that is about four times faster than the original.

Handledare: Stefan Engblom
Ämnesgranskare: Jarmo Rantakokko
Examinator: Olle Gällmo
IT 12 002
Tryckt av: Reprocentralen ITC

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Problem Statement And Motivation

One of the most important ways for contagious diseases to spread among cattle is through the transfer of cattle between different holdings. The verotoxin producing bacteria E. Coli O157:H7, VTEC O157:H7 (VTEC) can spread between holdings through cattle trade. VTEC is a zoonosis, i.e. it can transfer between animal and human. VTEC is transferred to humans through the intake of contaminated food or water. Children and elderly are more susceptible to VTEC infection and in humans the disease is then called EHEC [3]. The symptoms of EHEC can vary from mild to severe bloody diarrhea (often without fever) and a few cases also develop kidney failure, neurological symptoms or even death [9]. In a recent EHEC outbreak in Germany 46 people died — one of them a Swedish citizen [4]. The number of human cases contracting VTEC has increased since the mid nineties and therefore it is important to learn more about the spread of these diseases.

In order to investigate how VTEC infections can spread between farms by animal transfers a Monte-Carlo simulation model was developed [14]. In the simulation time is stepped one day at a time. Each farm (holding) has three categories of cattle (calf, young stock and adult) and neither of these categories affect the other. Each cattle is either susceptible for infection or infected and hence sheds the bacteria into the environment (fig. 1.1). The risk of an animal becoming infected differs between age groups and how much bacteria is present in the environment. When an animal becomes infected the length of the shedding period and how much bacteria the animal is going to shed each day is decided through randomized functions. Decline of bacteria in the environment is done each day and can vary e.g. depend on the season. The initial infection status of each individual are randomly assigned based on the spatial distribution of the infection according to previous studies of VTEC prevalence in Sweden. In the simulation, the infection status, age and location of each individual animal in the population is updated daily. All cattle births, transfers and slaughters are registered in a national database and an excerpt from this database over the period 2005-07-01 to 2008-12-31 were used as input in the simulation. The database contains more than 31 000 holdings, about 3.5 million cattle, and just over 8 million events (where almost 1 million of them are transfers between holdings). The algorithm of the model as it is implemented can be seen in alg. 1.1.

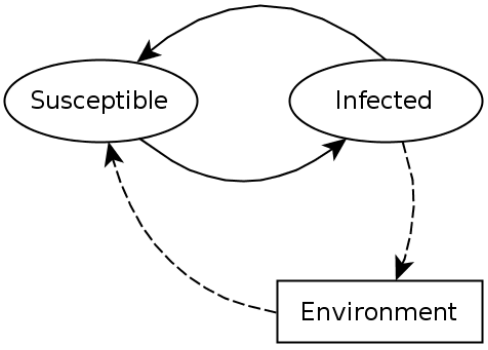


Fig. 1.1: Overview of an animal in the simulation.

```
1 foreach Simulation do
2     Reset data in animals and holdings from previous simulation
3     foreach Day do
4         Decide which season it is
5         foreach Holding do
6             Step simulation model one day
7             Save data if necessary
8         Handle events for this day
```

Alg. 1.1: The main algorithm.

The purpose of the program is to study the effects of different preventive measures (e.g. prevent holdings from transferring animals) on a multitude of modeled scenarios. To get statistical data for one specific scenario by default 500 simulations are run. The original implementation was not optimized for speed which resulted in run times of 8 hours or more to finish a single scenario.

To have some real world use of the program the run time has to be significantly shorter. The main objective of this thesis is to minimize the time needed to finish a test run and by doing so find out whether simulating spread of disease on an individual rather than on population level is feasible using a workstation anno 2011. Most if not all workstations and even laptops are equipped with multi-core CPUs which makes parallelization a prime candidate for speeding up programs. Different ways of optimizing data structures for good use of cache and minimizing main memory accesses are also investigated as it was discovered in an early stage that memory bandwidth was a bottleneck.

1.2 Delimitations

In this thesis we limit our investigations to minimizing the run time of the program on a single workstation class computer. There are ways of speeding up the execution using message passing and multiple computers e.g. run multiple simulations simultaneously, but we do not look into them in this thesis. We also do not look into alternative ways of storing data.

A less work intensive way of simulating the spread of disease is at the level of a single population. However, since data is available for transports, birth and death on individual animals, we investigate the possibility of simulating on the level of an animal. Such an approach facilitates setting the state of a specific animal in the beginning of the simulation in order to see how that specific animal will influence the outcome of a simulation.

Chapter 2

Design & Implementation

In this chapter the original implementation and its problems will be presented along with the methodology and tools used to analyze the program throughout the development process. We will also present the design goals and the avenues taken in order to try to fulfill them.

2.1 Original Implementation

In the original implementation there is a global vector of pointers to animal objects. For each holding, the animals are saved in a vector where each element points to another data structure (an std::map, which is usually implemented as some kind of tree [13]). Each element in the vector represents one of the six possible states of an animal. The data structure contains all animals currently in this state and an animal is represented by a pointer to an animal object. Furthermore, each event has a pointer to the animal object whom that event affects (fig. 2.1).

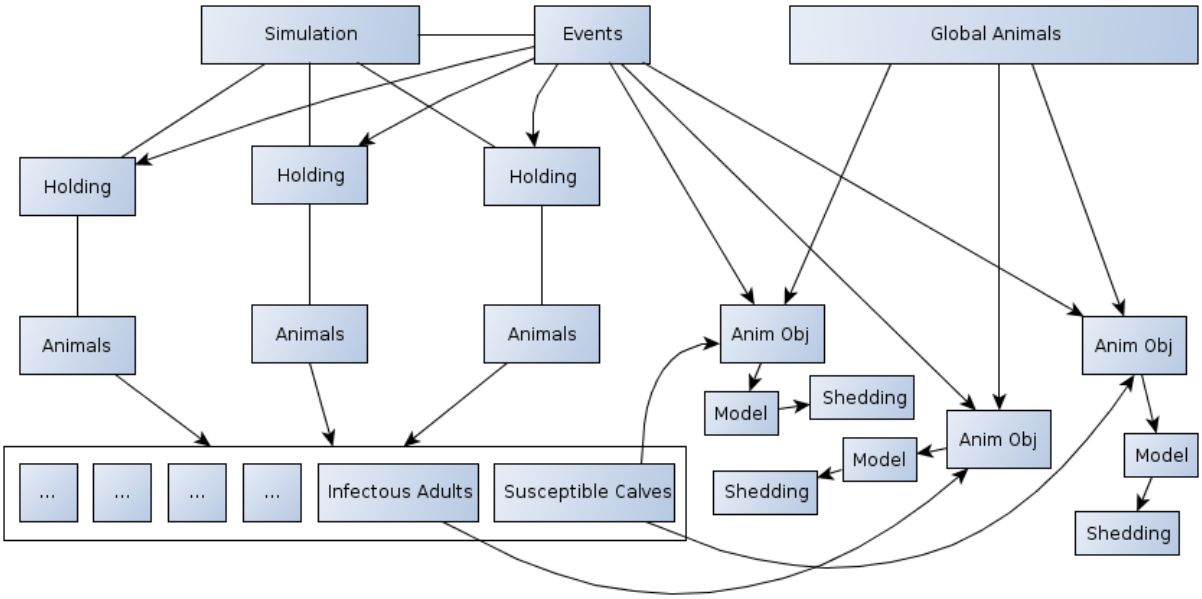


Fig. 2.1: Schematic overview of the original implementations kernel.

The design although flexible presents a problem for performance in regards to memory accesses. Every time data in an animal object is to be accessed two memory reads has to be done to find the memory address of the object. This means that the inner loop has to wait for address look ups from main memory and then wait for the object to be read from memory each iteration. To make the problem even worse the data that is to be manipulated most of the time resides in the model object which the animal object has a pointer to. On top of that the model objects which holds most of the data associated with an animal are not aligned in memory which makes it impossible for the prefetcher to predict coming iterations access patterns. To process an animal the inner loop goes through a chain of at least three pointers to get to the data that is to be read or manipulated. Of the pointers only the first in the chain to the six different animal state maps can likely be found in cache or registers for fast access. Even worse, when updating the prevalence of contagion in a holding the shedding

object has to be accessed which adds one more memory access and function lookup.

```
1  foreach Simulation do
2      Reset each animals status
3      Move animals to their respective holding
4      Expose animals to contagion
5      foreach Day do
6          foreach Holding do
7              Perform the step algorithm (alg. 2.2)
8              Save incidence data if present
9          Add and remove animals from simulation
10         Transfer animals between holdings
11         Age animals
```

Alg. 2.1: Overview of the main algorithm

```
1  Initialize a vector for animals that change state
2  Count number of susceptible and infectious animals
3  foreach Infectious animal do
4      Reduce number of days until susceptible
5      Increase number of days the animal has spent infectious
6      if Animal recovers then
7          Add animal to the change state vector
8  foreach Susceptible animal do
9      Expose animal to infectious dose
10     if Animal become infected then
11         Add animal to the change state vector
12 foreach Animal in the change state vector do
13     Perform appropriate action depending on state
14 Decrease prevalence of contagion depending on season
15 Increase prevalence of contagion depending on contaminated animals and season
```

Alg. 2.2: The algorithm performed each day for each holding.

Before each simulation, as can be seen in alg. 2.1, a reset of the holdings and animals is performed. This reset reverts the state of each animal to its initial state, move them to their initial holding and depending on initial prevalence of contagion at a holding decides using randomness if there are any infectious animals. After that, the season is decided by checking an array of days when to change the season (which can be given as an argument). Then the step algorithm (alg. 2.2) is performed for each holding and the number of susceptible and sick animals is saved if there are any sick animals in that holding. The reason for using the change state vector (alg. 2.2) is to avoid that an animal that were infectious, recovers and is contaminated again in the same step. Last the events (birth, death, age, transfer) are performed for each holding with events.

2.2 Design Goals

The design goals are:

- Make the code easy to read unless it adversely affects performance.
- Make the code easy to extend with new models and actions e.g. preventive measures.
- As low as possible run time on a workstation anno 2011

2.2.1 Objects And Their Performance

Object oriented programming have got some advantages compared to other programming techniques. Some of these advantages are reusability and using interfaces and virtual functions to easily switch a part of an object without rewriting a lot of code. In our case the animal object holds a pointer to a model object (fig. 2.1) and thus in theory you could switch the model simulated by simply telling the animal constructor to instantiate a different kind of model. Both the animal and model objects are instantiated at the bottom of a fairly long inheritance chain. In animal objects non virtual functions in the interface call virtual functions sometimes all the way down to a bottom object. This causes the compiler to never inline the function call even if the function is a one line getter i.e. is a really good candidate for inlining. This is the main reason why we merged the animal and the epistate objects in the new architecture discussed in section 2.5.2. To further speed up execution we also made some fields of the new animal object public to remove the use of a function completely. Both merging the data into one object and making some fields available to owners of the new class facilitates inlining not only in the object itself but also on the higher levels. This can have a dramatically positive effect on performance [1]. In the parts of the program that have less of an impact on performance, readability and ease of use remained our highest priority.

2.3 First Naive Parallelization

In order to see how well the program handled parallelization we investigated two different naive parallelizations. The first version was a parallelization over the holdings, that is, for each day the holdings were divided between different threads. In the second version parallelization was done over the animals in a holding. As can be seen in alg. 2.1 the first version looks more promising since each holding can be individually calculated and it is possible that there is enough work to be done there to offset the overhead of creating threads.

Indeed, after trying out both of these versions, the first version (parallelization over holdings) proved to be the best by far. Unfortunately we did not get good speed up(see fig.3.1(a)) and decided to investigate where the time was spent and how the design could be improved for parallelization.

2.4 Methods And Tools Used For Analysis

The work flow of this thesis has been to use profilers to find where and why the simulator were spending time and then optimizing those parts for parallelism and speed. The tools mainly used for this were Valgrind [11] and the profiler in Microsoft Visual Studio [10]. Intel Parallel Amplifier 2011 [12] were used to profile the multi threaded code and see where the threads lock and/or wait for resources. To be able to efficiently benchmark and compare optimization strategies a python script was developed for running tests. The script produced run times and Latex [7] compatible speed up and run time graphs.

2.4.1 Valgrind

Valgrind in essence is a virtual machine which can run any binary. When the program is run Valgrind insert its instrumentation. Valgrind can be used as a memory checker but in this project it has mainly been used to profile cache utilization and branch prediction. Figure 2.2 is an example screen shot from KCachegrind [6] — the tool used to analyze data from Valgrind. In figure 2.2 you can see that a specific switch statement is responsible for more than 29% of the misspredicted indirect branches. As a result of this we looked into splitting up animals into one vector for each state thus removing the switch completely.

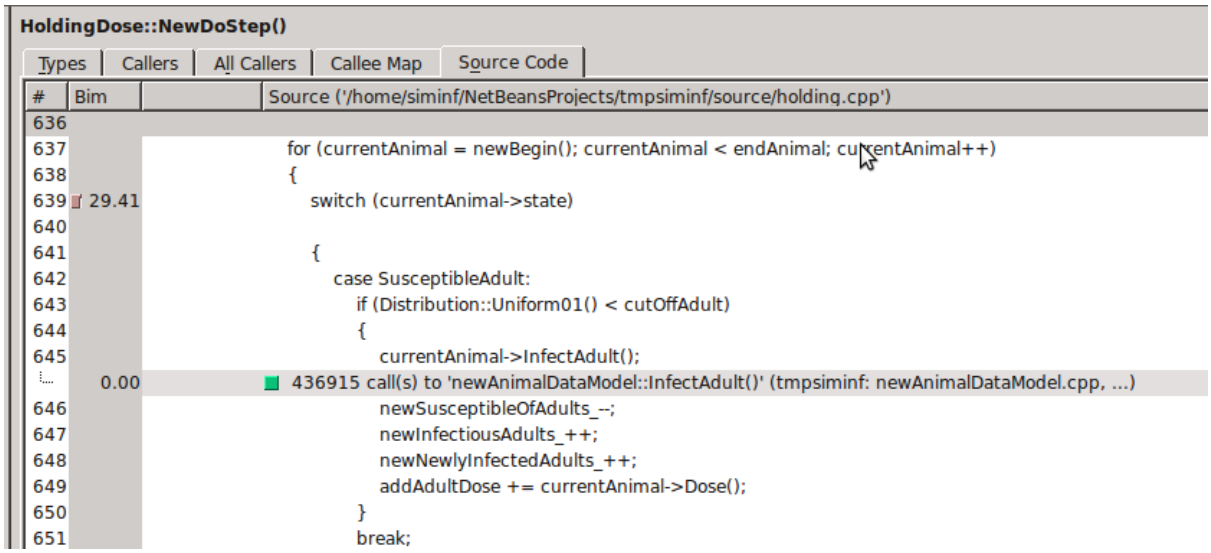


Fig. 2.2: Example screen shot from Kcachegrind.

2.4.2 MS Visual Studio profiler

Microsoft's profiler were mainly used in order to see how much time were spent inside functions. We used the sampler part of the profiler. A sampler runs the program for a certain amount of time and then it checks which function is running right now. It then saves that information and continues until the program is terminated. When this is done visual studio allows you to look at that data in a graphical manner in order to see how many times a function were running when the profiler performed the check. Fig. 2.3 is an example of data from the profiler. The marked operation is the operation that find the next animal in the vector and it should be noted that more time is spent there than

is spent in the dose-response computation above the mark. Bacause of this we looked into improving memory accesses as explained in section 2.5.2.

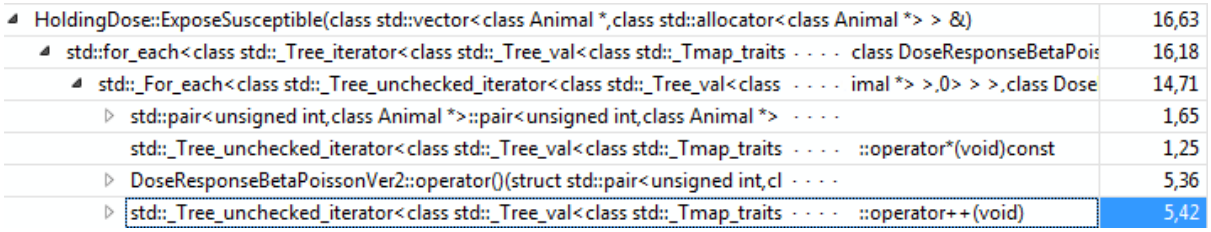


Fig. 2.3: Example of data from Microsoft Visual Studio profiler.

2.4.3 Intel Parallel Amplifier

Intel's profiler were used in order to find out where threads are locking and/or waiting for resources. This can be seen by looking at the data generated from the profiler. The profiler shows how good the utilization of a function has been during a certain time by showing different colors depending on how much the CPU has been working during that time.

2.4.4 Intel Parallel Advisor

Intel has got an advisor that help you by showing where you can parallelize and what can be done in order to increase speedup. We did not use this extensively, an exception being in order to see if the position where we chose to parallelize were any good and get some hints on what to do. In the end, it did not do much for us since we already knew all that it had to tell.

2.5 Design Choices

In this section we describe the optimizations we tried in order to make the program run faster. Parallelization was a big part of the speed up effort. We begin by discussing which parts of the code is suitable for parallelization and continue with optimizations that are meant to increase the gain from parallelizing the program.

2.5.1 Where To Parallelize

The tool used to parallelize the original implementation were OpenMP [2, 5]. The main reason for using OpenMP is its ease of use and that parallelizing with OpenMP does not consume a lot of time. The most important reason however was that we judged other code optimization to be more important and we could not think of a way to increase performance by using something more versatile (e.g. POSIX Threads [5]).

2.5.1.1 Parallelization Over Simulations

Most of the data a simulation uses can not be shared with another simulation even if the starting conditions are identical. Since the code does a lot of reads and writes to

main memory there is a risk of memory bandwidth being a bottleneck. Because of this, parallelization over simulations is not a very promising idea for thread level parallelization. Parallelization over simulations using message passing is in principle not a bad idea but that is outside the scope of this thesis.

2.5.1.2 Parallelization Over Holdings

Since a holding during a specific time discretization is isolated from the other holdings it is a trivially parallelizable unit as long as you synchronize with the other holdings when global events happen that affect the specific holding. There is a problem with load balancing in this level of parallelization in that there can be a big difference between the number of animals each holding has. This is the level we chose to parallelize over.

2.5.1.3 Parallelization Over Animals

When parallelizing over Animals the overhead of creating threads are higher than the gain of running multiple threads. The relatively small amount of work to be done for each animal for each step is to blame for that . Even if one thread is given a chunk of several animals to work on the amount of work is too small to make it worthwhile. We therefore did not further investigate this strategy.

2.5.2 Merging Animal And Model Object

Something important to consider when designing a program for parallelization is the size of the data that is going to be used and make sure that as much as possible of that data is used [5]. Fetching unnecessary data occupies resources that could be used for something else. The animal object in the original implementation (see fig. 2.4) had a few variables that could be removed and a pointer to the epidemiological model which is used whenever we work on an infectious animal.

In order to solve these problems we decided that we should merge the epidemiological model with the animal object. While merging the objects we tried to reduce the size of the new object as much as possible by removing variables that could be stored in some other way. We also defined a new type (AnimalState) containing all of the possible states of an animal in order to have an easier time debugging our code and make it easier to read.

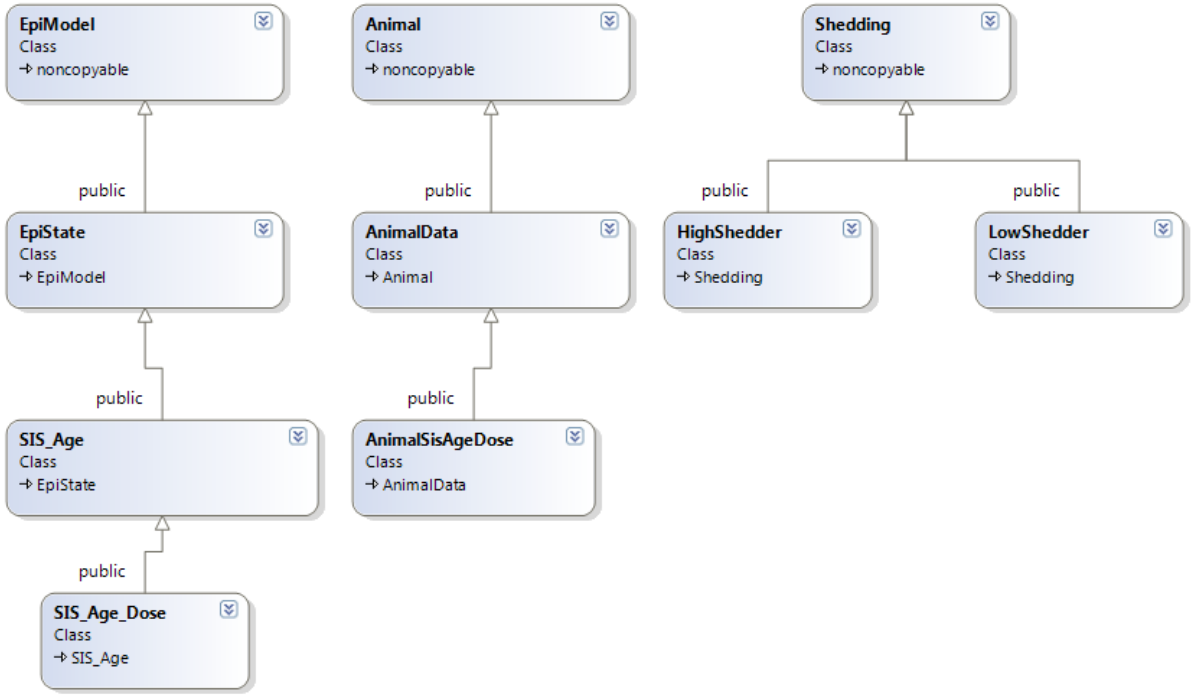


Fig. 2.4: Class diagram of the original animal object.

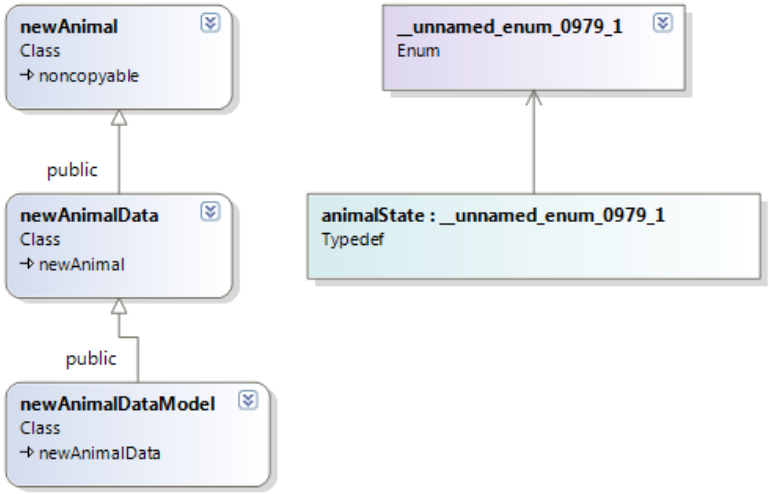


Fig. 2.5: Class diagram of the merged object.

2.5.3 Moving Animals To Holdings

In the original implementation the animals could be found all over the memory which is not ideal for performance. When parallelizing we want memory accesses to be as linear as possible in order to allow the prefetcher to fetch data to cache efficiently [5]. Since we want data to be stored and accessed sequentially in memory we decided that we should move the animals into their respective holding. To store the animals the vector data structure in the C++ Standard Template Library was chosen.

The behavior of the vector class is similar to a C-array in that it store data sequentially and allows random access, but it also has other features e.g. automatic resizing when necessary. There is however a potentially big cost when using vectors, and that has to do with removing objects in the beginning or middle of the vector. When removing an object in the beginning or middle of a vector, the vector need to fill the gap. A vector fills a gap by moving all objects with a higher index to a lower index by a number of steps equal to the number of removed elements. To circumvent this cost we always swap the object to be removed with the last object in the vector and then remove from the last position in the vector. By doing so the cost of removing an animal is negligible.

2.5.4 Separating Animals Into Different Vectors

One of the problems with having one vector where all animals are stored is that both susceptible and infectious animals are grouped together. Since we perform different actions on infectious and susceptible animals we split them into two different vectors, one for infectious animals and one for susceptible animals. This also allowed us to remove some branches, in particular the one checking if an animal is susceptible or not. Since animals are stored in the order they arrive this branch were quite random which resulted in bad branch prediction results as observed using Valgrind [11]. Good branch prediction in code that is run repeatedly and often with little work in between can be very important to get good performance [5].

In order to further reduce the size of the animals and at the same time remove some more branches we decided to remove the state variable. There are six different states of an animal and thus the information stored in this variable can be kept by using six vectors (one for each state). Splitting animals into six different vectors could potentially help performance when searching for animals by deciding what order they should be searched in. An overview of the new data structure can bee seen in figure 2.6.

One small negative side of splitting animals into six vectors is that for each vector prefetching has to restart. Another drawback that we now have to consider is the case where animals change state when they should not e.g. infectious $\rightarrow$ susceptible $\rightarrow$ infectious. Since we go through the animals one state at a time we solved this issue by using a temporary vector where we stored the animals that changed state and added them to the appropriate vector when animals of that state were done. An example is when an infectious animal recovers. The animal is added to the temporary vector (as a susceptible animal) and is added to the vector of susceptible animals when we are done contaminating the susceptible animals.

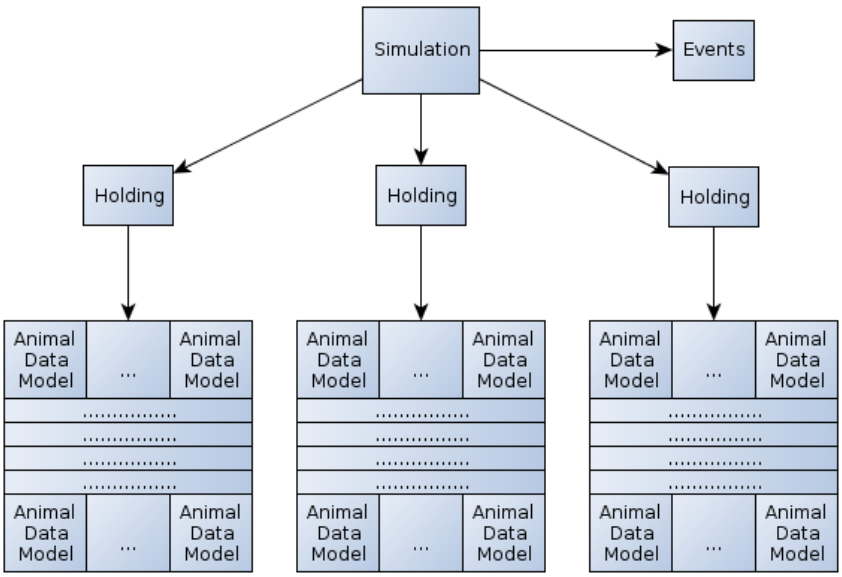


Fig. 2.6: Overview of the new data structure implementation

2.5.5 Data Storage

Since sqlite is used to save data and it does not support multiple threads executing writes to the database at the same time we have a serial code segment for each holding. It is well known that having serial code segments will inhibit the possibility of speedup [2, p. 173-174] using multiple threads. The two obvious ways of handling this problem are using one database per thread or using a lock to ensure only one thread at a time writes to the database. Both methods were investigated and found to be good in different scenarios. If more than one database is used they have to be merged unless the tools used to analyze and manipulate the results are also rewritten to use multiple databases.

2.5.5.1 Single Database With A More Fine Grained Lock

If one global sqlite database is used only a single thread at a time should be allowed to write to the database. This can be accomplished by using a lock. In the method writing to the database there is a check to see if there are any sick animals and, if so, a write to the database is performed. In the naive parallelization we used a lock for this entire method even for holdings that do not write anything to the database. To minimize the overhead introduced by the lock we moved the lock into the code block that writes to the database.

2.5.5.2 One Database Per Thread

In order to avoid using a lock we investigated using one database per thread and optionally merging them after the simulations were done. The merging of two or more databases is done without parallelization. The cost for merging databases does not seem to be tied to the number of databases being merged in the cases we have investigated as can be seen in tables 3.3, 3.4 and 3.5. The cost of merging databases were higher when using windows compared to Ubuntu on the system we ran tests on. The reasons for the

relatively large difference between windows and Linux were not investigated.

2.5.6 Fast Forward

During one day the holdings are isolated from each other and there is an average of almost 150 event days for each holding during a period of 1280 days. This means that there is on average almost 9 days between the events in a holding. This gave us the idea to 'fast forward' to the day before the next event occurs in one go and by doing so hopefully use the memory cache structure more effectively. If a fast forward end at the last day of the simulation the holding is deactivated. This optimization introduces a possible load imbalance since the number of days between events for a holding varies a lot.

2.5.7 Deactivating Holdings

Holdings start out as deactivated if they have no infectious animals, and are activated if they receive an infectious animal through a transfer. In an inactive holding the step algorithm 2.2 is skipped, but events like aging and transfers do occur. The holdings were never deactivated again after this step.

An optimization were investigated where a holding is deactivated when there are no infectious animals and the risk of being contaminated is below some threshold. This threshold can be given to the simulator as a parameter and defaults to 0.1. Testing showed that a threshold large enough to make a substantial difference on run time also had a non negligible effect on the outcome of the simulation. This led to the removal of this optimization in the later stages of the development.

2.5.8 New Random Number Generator

After optimizing the other parts of the program, the Random Number Generator (RNG) function caught our attention by using a relatively large amount of the CPU time. The RNG that dominated the cost of generating random numbers were the RNG for generating uniformly distributed numbers between 0.0 and 1.0 called in the holding and animal objects. The given implementation using a single static randomization object was not modified to be thread safe and was associated with costs for calling methods finding the object etc. A thread safe and faster way of generating random numbers using a pure C macro [8] were investigated.

The macros are made thread safe using different methods for the animals and the holdings. Since we strive to keep the size of the animal objects small a global set of RNG variables protected by a lock is used for all animal objects. In the holdings where size is not a big issue we use local RNG variables in each holding object instead of the global variables used for animals. This method works since only one thread at a time is using a specific holding.

The RNGs in the current implementation are seeded with fixed seed values in order to have repeatable results, of course this should be fixed before using the program in a real setting.

Chapter 3

Results

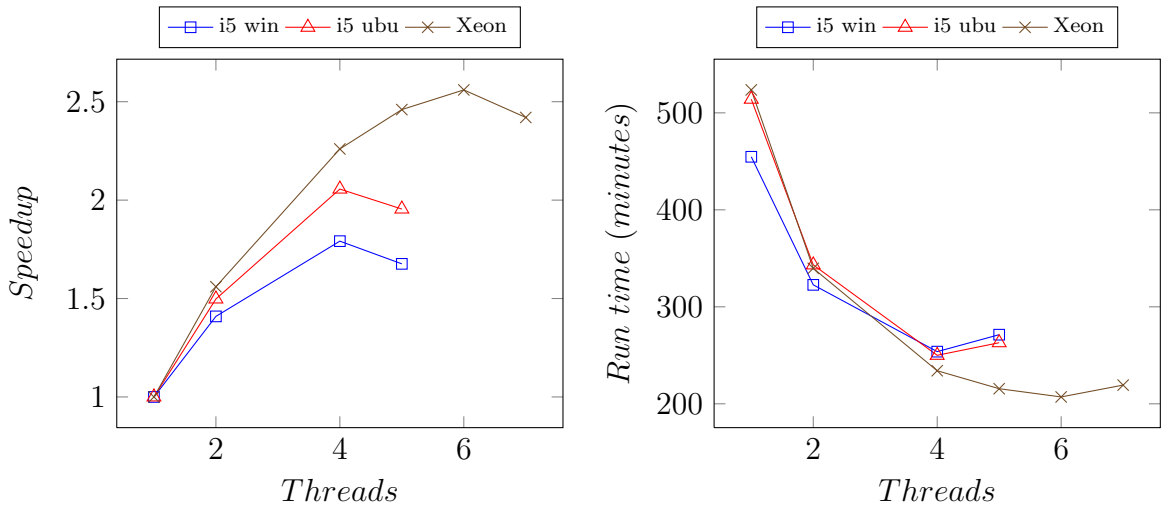
In this chapter we discuss speedup and run times for the different optimizations we implemented. We will begin by discussing the first naive parallelization and after that the fastest version we implemented. Then we will continue with the new data structure, the different configurations with the new RNG, single vs multiple databases and finally increasing the number of writes to the database. For everything except the naive implementation the speedup is based on the serial run time of the version with a new data structure. Three different systems are discussed here: *i5 ubu*, *i5 win* and *Xeon*. The information regarding these systems can be found in appendix A. As mentioned in section 2.5.1.2 there is a possibility of load imbalance. After testing different scheduling techniques in OpenMP (with only a slight increase/decrease in performance) we decided to use dynamic scheduling with a chunk size of 20 holdings.

3.1 First Naive Parallelization

Here we compare the first naive implementation with the three different systems. There are a few interesting points to pay attention to. One of them is that the serial run time of *Xeon* and *i5 ubu* is considerably higher than the serial run time for *i5 win*. The speedup for *i5 win* however is worse than for *Xeon* and *i5 ubu*. This is because the serial run time of *i5 win* is lower than for the other two systems, but the run time is almost the same when running holdings in parallel (fig. 3.1(b)). Also, as discussed earlier in section 2.3 the speedup is not very good as can be seen in fig. 3.1(a).

Threads	Run time i5 win	Run time i5 ubu	Run time Xeon
1	07:34:42	08:34:04	08:50:26
2	05:22:39	05:43:14	05:39:42
4	04:13:48	04:10:01	03:54:05
5	04:31:16	04:22:59	03:35:54
6			03:27:05
7			03:39:18

Tab. 3.1: Naive Run times



(a) Speedup

(b) Run time

Fig. 3.1: Naive parallelization

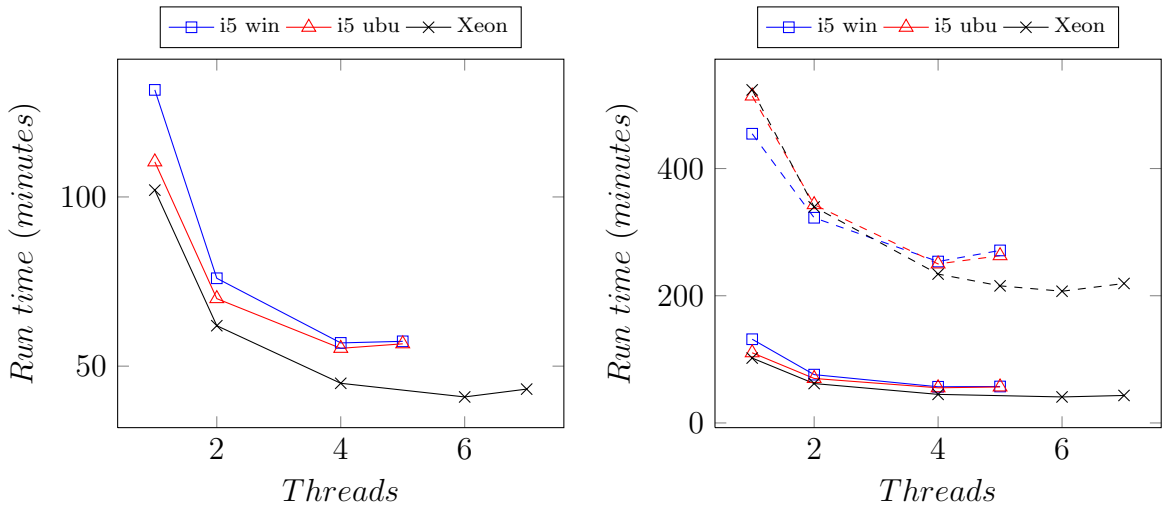
3.2 Final Results

3.2.1 Fastest Times

Here we compare the naive implementation with the fastest implementation we found. The fastest version is the new RNG in the holding object explained in section 2.5.8. Remember that in the naive implementation the best run time is 4 ± 0.5 hours. If we compare this time with the time of the fastest version — which is below one hour — we can reasonably claim that the new version is about four times faster (see fig. 3.2). Something that should be noted is that the speedup of the fastest version is actually better than the speedup of the first naive implementation, but still not impressing (see 3.3). The dashed lines in the graphs are the naive versions.

Threads	Run time i5 win	Run time i5 ubu	Run time Xeon
1	02:11:38	01:50:22	01:42:01
2	01:15:58	01:09:57	01:01:57
4	00:56:52	00:55:16	00:44:56
5	00:57:20	00:56:36	
6			00:40:54
7			00:43:13
Merge	00:24:07	00:07:18	00:02:52
Including Merge	01:20:59	01:02:34	00:43:46
Single DB	01:17:31	01:02:05	00:48:50

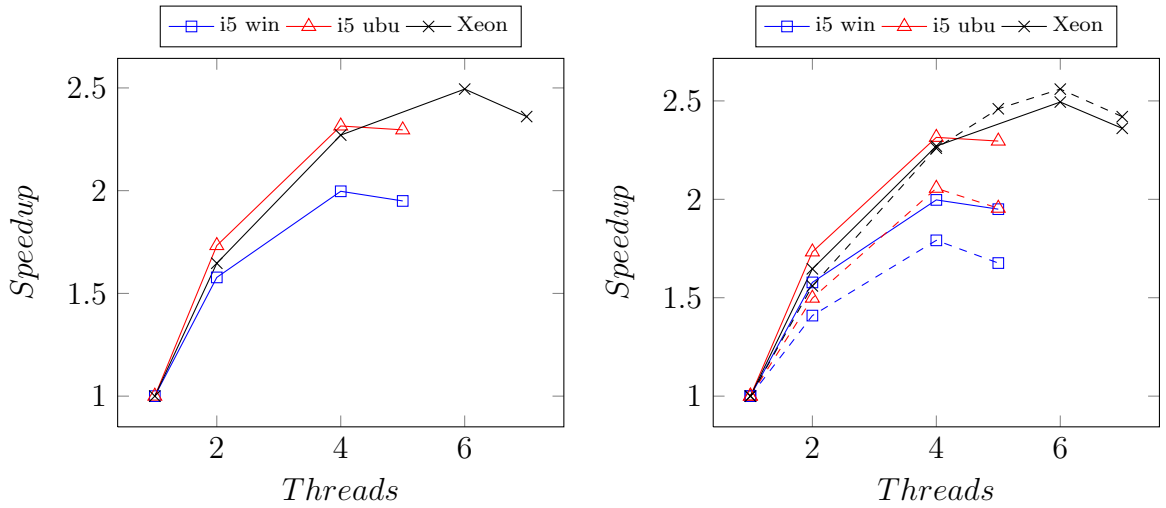
Tab. 3.2: Fastest Times



(a) Without Naive Time

(b) With Naive Time

Fig. 3.2: Best times excluding merge.



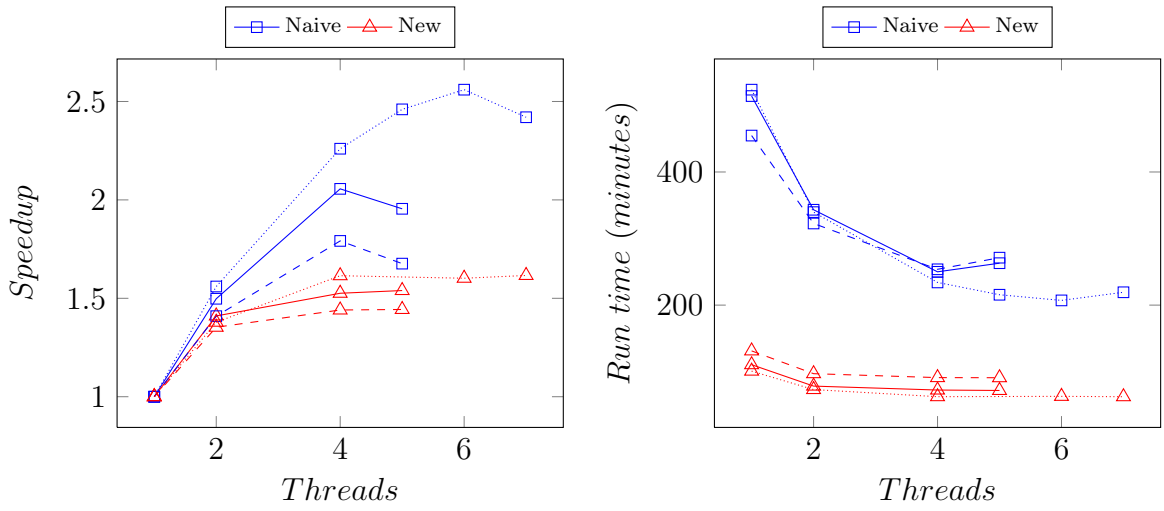
(a) Without Naive Speedup

(b) With Naive Speedup

Fig. 3.3: Speedup excluding merge.

3.2.2 New Data Structure

This is the version where the new data structure is implemented (see section 2.5.2) but there is no other optional optimization active. This version alone is almost four times faster than the naive implementation (fig. 3.4(b)). The speed up however is significantly lower when using more than two threads (fig. 3.4(a)). The reason for this is that there are more threads waiting for memory access and not enough calculations to be done in order to mask the time spent waiting for the memory. In the graphs below a solid line represents *i5 ubu*, a dashed line represents *i5 win* and a dotted line represents *Xeon*.



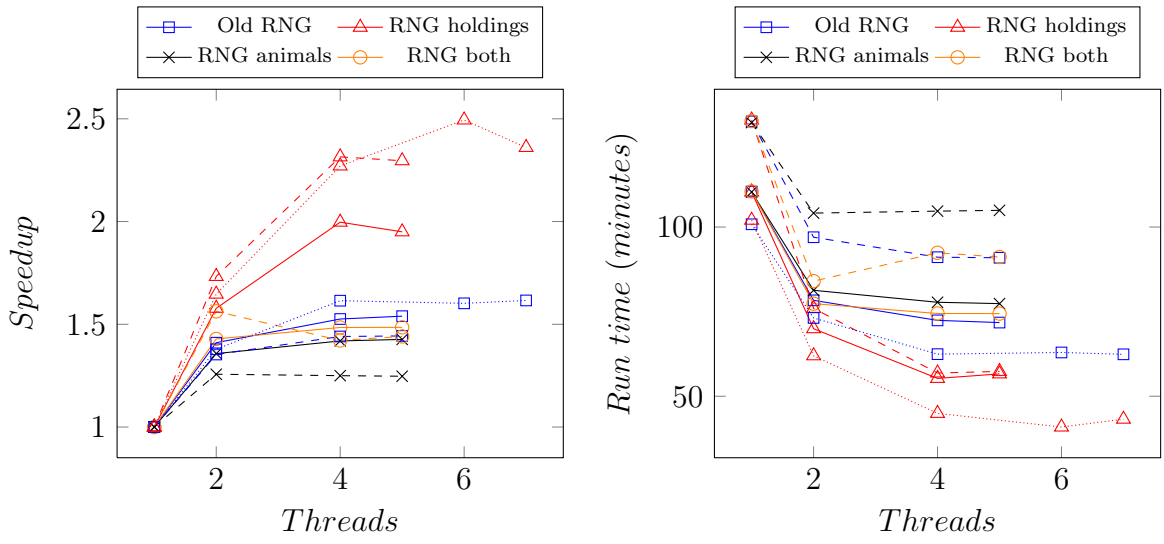
(a) Speedup

(b) Run time

Fig. 3.4: New data structure

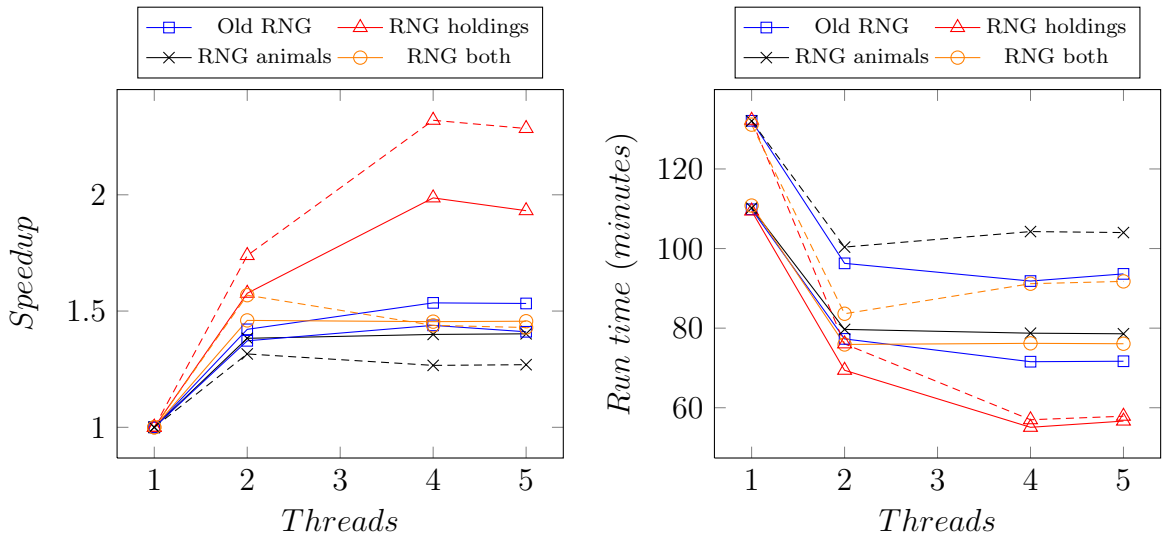
3.2.3 Marsaglia RNG

In this section we present the different configurations of the new RNG discussed in section 2.5.8. As can be seen below the version where we use the new RNG in the holding objects is the fastest of them. This is because that is the place where it is most frequently used and not having to do any function look ups reduces the time spent there. The version where we changed RNG in the animal objects is worse than the old RNG which can be explained by the use of locks and that there is a high probability that more than one thread will handle a sick animal. This slowdown is slightly mitigated when we combine the different versions with each other. In the graphs below a solid line represents *i5 ubu*, a dashed line represents *i5 win* and a dotted line represents *Xeon*.



(a) Speedup (b) Run time

Fig. 3.5: RNG without fast forward



(a) Speedup (b) Run time

Fig. 3.6: RNG with fast forward

3.2.4 Single vs Multiple Databases

In this section we compare the times of using a single database with using multiple databases as discussed in section 2.5.5. Recall that the problem with using locks is that everything risks being serialized. If we look in the tables 3.3, 3.4, 3.5 we see that this is not a problem if we have to merge since the total run time when merging is almost the same. Something to note though, is that when using Ubuntu, the program is seriously slowed down (in part due to threads waiting for locks) when using more threads than

available cores for the computer (see fig. 3.7 and tab. 3.4). In the graphs below a solid line represents *i5 ubu* and a dashed line represents *i5 win*.

(a) Multiple databases

Threads	Run time	Merge time
1	01:41:47	
2	01:01:57	00:02:52
4	00:44:56	00:02:52
6	00:40:54	00:02:52

(b) Single database

Threads	Run time
1	01:41:47
2	01:05:15
4	00:51:06
6	00:48:50

Tab. 3.3: Run times *Xeon* with multiple databases and merge vs a single database

(a) Multiple databases

Threads	Run time	Merge time
1	01:50:22	
2	01:09:57	00:07:18
4	00:55:16	00:07:18
5	00:56:36	00:07:18

(b) Single database

Threads	Run time
1	01:50:22
2	01:14:14
4	01:02:05
5	01:46:14

Tab. 3.4: Run times *i5 ubu* with multiple databases and merge vs a single database

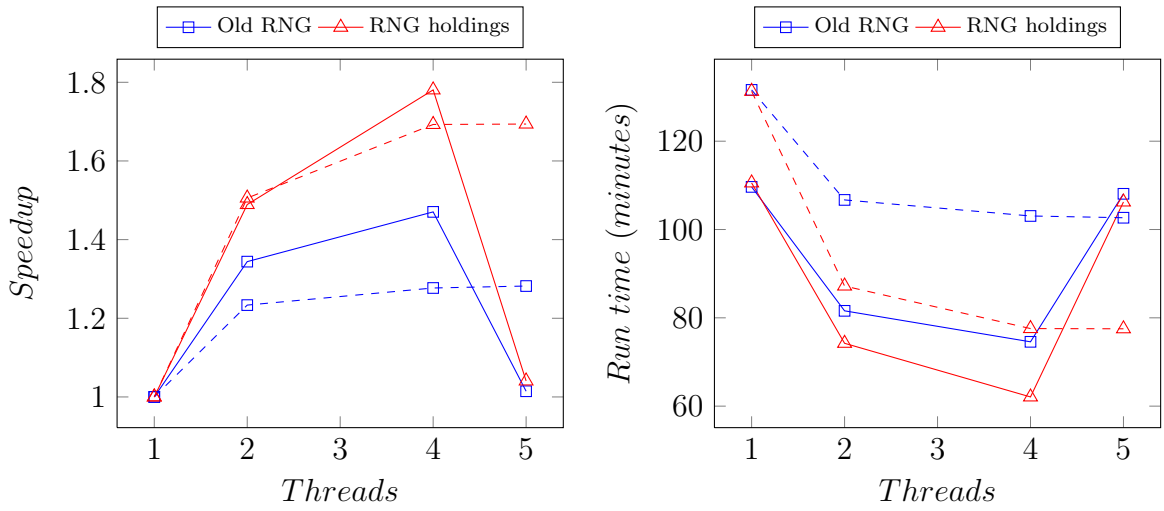
(a) Multiple databases

Threads	Run time	Merge time
1	02:11:18	
2	01:15:58	00:24:07
4	00:56:52	00:24:07
5	00:57:20	00:24:07

(b) Single database

Threads	Run time
1	02:11:18
2	01:27:12
4	01:17:34
5	01:17:31

Tab. 3.5: Run times *i5 win* with multiple databases and merge vs a single database



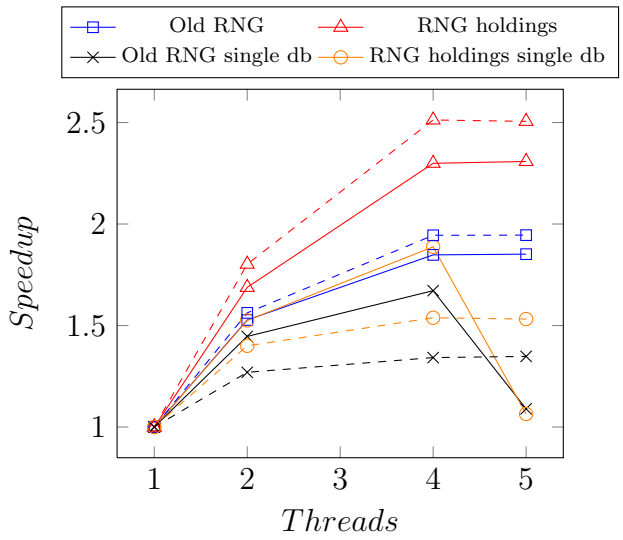
(a) Speedup

(b) Run time

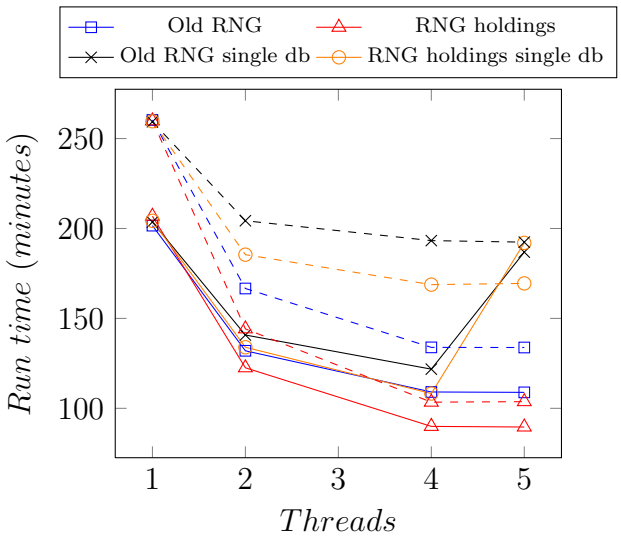
Fig. 3.7: Using a single database

3.2.5 Increased Writes To Data Storage

In this last section we wanted to check how the implementation handles more writes to the database. To force more data to be generated we doubled the probability of a newly contaminated animal becoming a highshedder. When an animal is a highshedder it secretes many times more bacteria into the environment causing more animals to become infectious and thus more writes to the database. As can be seen in figs. 3.8(a), 3.8(b), 3.9(a), 3.9(b) the single database version did not handle it too well, but it is still the fastest version since merging (especially in windows) represents a large amount of the run time. There is one case where the multiple database is still faster (including merge) than the single version and that is when the new RNG is active in the holdings in Ubuntu. In the graphs below a solid line represents *i5 ubu* and a dashed line represents *i5 win*.

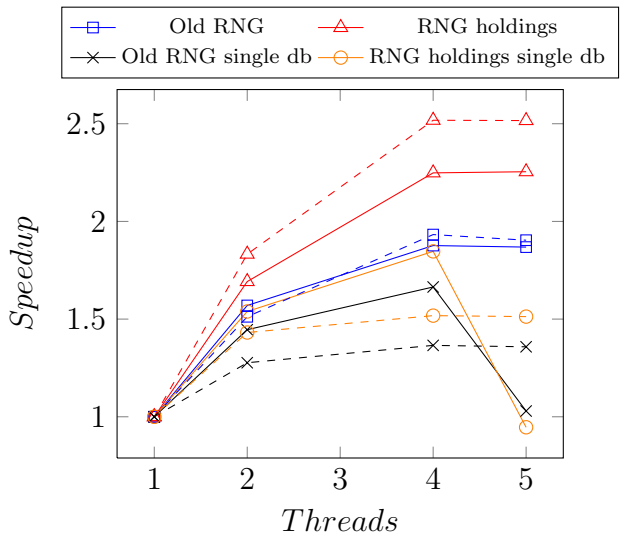


(a) Speedup

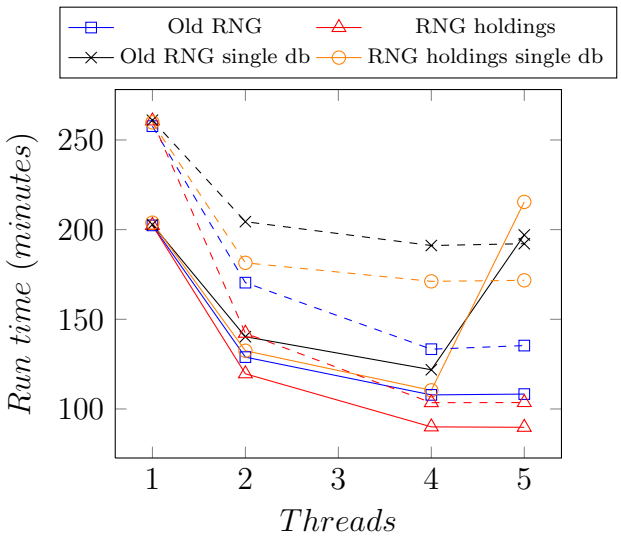


(b) Run time

Fig. 3.8: Increased writes without fast forward



(a) Speedup



(b) Run time

Fig. 3.9: Increased writes with fast forward

Chapter 4

Conclusions

4.1 Conclusion

The main objective of this thesis has been to minimize the time needed to finish a given test run and by doing so find out whether simulating spread of disease on an individual rather than on a population level is feasible using a workstation anno 2011. We were given a working copy of a program that simulated the spread of disease among animals on an individual level but the run time was too long in order to use it effectively.

We started out by parallelizing the original implementation and continued with analyzing the program. After analyzing the program we decided to change the data structure of the animals since there were a lot of memory accesses when handling updates and operations on individual animal objects. This new data structure decreased the run time by several hours showing us that the run time goal were attainable. Then we changed the RNG and lowered the run time even more to the final results that can be seen in fig. 3.2.

We are confident that this new run time makes it possible to use the simulator in research on a workstation anno 2011. Particularly so if you consider that it is possible to run it distributively with different parameters on different computers. It is our hope that in the future a program designed on our findings can be used to recommend actions to be taken to minimize the spread of disease among animals and in doing so lessen the risk of human illness or death.

4.2 Discussion

The work flow of this thesis has been to use profilers (section 2.4) to find where and why the simulator were spending time and then optimizing those parts for parallelism and speed.

The main part of the time was spent choosing the right data structures. With a proper representation of data, the algorithm and its parallelization was not that difficult. During the coding, when it was possible, changes were introduced in small incremental steps. By leaving the old data structures in the code, comparisons could be made to make sure that new data structures held the same information as the old one. Sometimes however it was not possible to keep the bugs out: a day and a half were spent looking for an erroneous ! in an if-statement.

While we managed to do a lot there were two things in particular we would have liked to try given more time. The first was to remove the animal objects all together (section 4.3.1) to maximize the performance of the memory subsystem. The second was to move the handling of most events into the holding object (section 4.3.2).

The difference between the run times of *Xeon* and *i5 win/ubu* are not that big i.e. the algorithm does not scale well with increasing number of available cores. This indicates that the bottleneck is not raw CPU power but rather giving the CPU data to work on. In our tests the difference in run time between using a single database and multiple databases (including the merge-time) is almost the same. Merging databases takes time linearly proportional to the size of the data on the data sets we have run it on. However, if the merge-time is excluded in the comparison, using multiple databases is faster than a single database. If the programs used to generate statistical data is rewritten to use several databases (sqlite has got good support for this) a lot of time can be saved (especially if

the operating system used is windows). There is also the problem that if a single database is used only one thread at a time can write to the database. This could seriously harm the performance if a lot of writing has to be done.

4.3 Future Work

4.3.1 Animal Object Vectors vs Vectors of Animal Data

Instead of using vectors of animal objects in holdings a more cache friendly way would be to have several vectors holding the data of the objects and having the index specify the animal. One way of doing this is using an *animals* object which hold the 18 different vectors needed to separate by state and hold the 3 data fields.

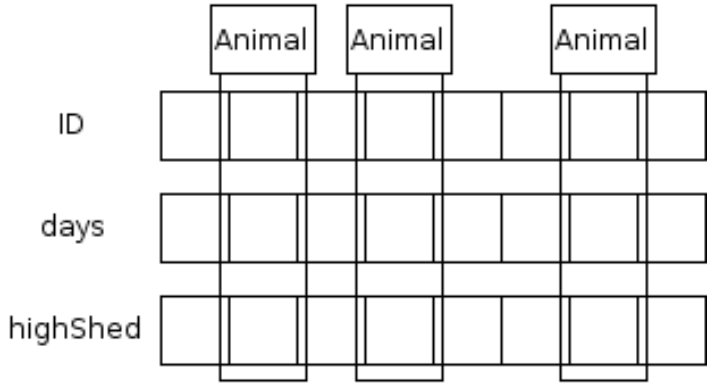


Fig. 4.1: Storing animal data without any animal objects

When animal data is accessed and or modified in the animal loops (alg. 2.2) we are never interested in the ID of the animal but since they lie next to each other in memory they will needlessly be put into cache memory. Conversely when searching for an animal to apply an event to it we are only interested in the animal ID which gives even worse cache usage. Splitting animals into vectors will help us avoid capacity misses in the cache. A special boolean vector type is available in C++ which means that each highshed boolean can be stored as a single bit, when the boolean type is used in an object the object is padded to align in memory thus wasting space. Assuming a memory bit width of 128 bits you can get the highshed boolean for 128 sick animals in one single main memory read. Since the prefetcher of a modern CPU have no problem following multiple access patterns at the same time when data is requested for a sick animal there is a good chance it will already be present in cache. One negative side with this optimization is that it makes the code more complex. In particular moving an animal from age group to age group or between holdings would become a more complex operation.

4.3.2 Move Events To The Holding Object

Of the four events Birth, Aging, Death and Transfer only one affects another holding than the one the animal is currently in, namely Transfer. This makes it possible to move Birth, Aging and Death into the holdings. This means that a lot of events can be processed in

parallel and this would also make the number of days the Fast Forward optimization (section 2.5.6) can "fast forward" longer. This will most likely give an increase in speed when using the Fast Forward optimization. Implementing this optimization would make the serial part of the program smaller which will increase the theoretically possible speedup [2, p. 173-174].

4.3.3 Better Implementation of Thread Safety For RNG

A more elegant and possibly more performant way of making the Marsaglia RNG thread safe is to have private RNG variables tied to each thread. This way you could have one set of global RNG variables for each thread instead of one set in each holding and a global set for animals.

4.3.4 Parallelizing Over Simulations

Most of the data that a simulation use cannot be shared between different simulations. This means that running multiple simulations on the same CPU in different threads will tax the memory system heavily and increase the risk of cache misses. Using message passing between different systems however would remove such disadvantages. This could also speed up simulating a series of scenarios by lowering the number of simulations run on one computer. Thread level parallelization could be investigated here but it is not something that we recommend.

4.4 Acknowledgment

We would like to extend our sincere gratitude to our supervisor Stefan Engblom for giving us pointers during the design and implementation and helping us to write this thesis. We would also like to extend our sincere gratitude to our project owner Stefan Widgren for giving us this opportunity and helping us to write this thesis. Last but definitely not least we would like to thank the coffee and tea makers for making this thesis and the code possible to produce.

Bibliography

[1] Dov Bulka and Mayhew David. *Efficient C++*. Addison-Wesley, 1999.

[2] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[3] European Centre for Disease Prevention and Control. Basic facts on escherichia coli (e.coli). `http://ecdc.europa.eu/en/healthtopics/escherichia_coli/basic_facts/Pages/basic_facts.aspx`, November 2011.

[4] European Centre for Disease Prevention and Control. Ecdc reviews - shiga toxin-producin e. coli (stec): Update... `http://ecdc.europa.eu/en/activities/sciadvice/Lists/ECDC%20Reviews/ECDC_DispForm.aspx?ID=1166`, November 2011.

[5] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[6] Weidendorfer Josef. Kcachegrind. `http://kcachegrind.sourceforge.net/`, November 2011.

[7] Latex-project. Latex – a document preparation system. `http://www.latex-project.org/`, November 2011.

[8] George Marsaglia. Random numbers for C: The END? `http://groups.google.com/group/sci.math.num-analysis/browse_thread/thread/ca8682a4658a124d/`, October 2011.

[9] Paul S. Mead and Patricia M. Griffin. Escherichia coli o157:h7. *The Lancet*, 352(9135):1207 – 1212, 1998.

[10] Microsoft. Microsoft visual studio. `http://www.microsoft.com/visualstudio/en-us`, October 2011.

[11] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42:89–100, June 2007.

[12] Intel® Software Network. Parallel Amplifier 2011 from Intel - Intel® Software Network. `http://software.intel.com/en-us/articles/intel-parallel-amplifier/`, Oct 2011.

[13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[14] Stefan Widgren and Jenny Frössling. Spatio-temporal evaluation of cattle trade in Sweden: description of a grid network visualization technique. *Geospatial Health 5*, 2010.

Appendices

A System Information

A.1 i5 win/ubu

Intel® Core™ i5-750 Processor (2.66 GHz, 4 cores, 8 MB total cache, 2 memory channels)
4 GB 1333 MHz DDR3 SDRAM
HDD Seagate® Barracuda® 7200.12 RPM (st3250318as) with ext4 file system for Ubuntu
HDD Seagate® 7200 RPM (VB0160EAVEQ) with NTFS(compressed) for Windows
Windows 7 SP1 and Ubuntu 11.04-Natty
In figures and tables these configurations are called *i5 win* or *i5 ubu* depending on the operating system used (Windows or Ubuntu) for the execution.

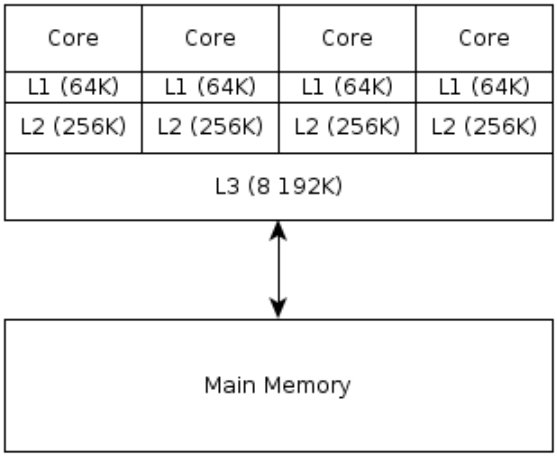


Fig. A-1: Cache architecture of *i5 win* and *i5 ubu*.

A.2 Xeon

Intel® Xeon™ W3680 Processor (3.33 GHz, 6 cores, 12 MB total cache, 3 memory channels)
24 GB 1333 MHz DDR3 SDRAM
HDD Hitachi™ Deskstar 7K1000.C (HDS721010CLA332)with ext4 file system
Ubuntu 10.10-Maverick
In figures and tables this configuration is called *Xeon*.

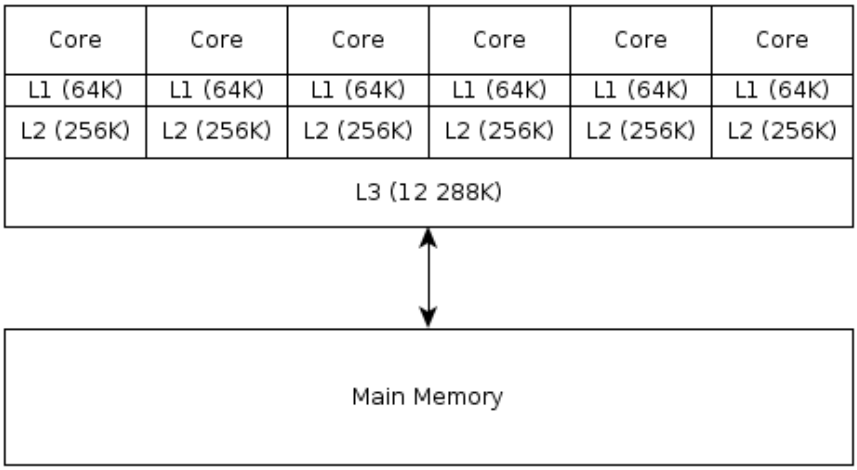


Fig. A-2: Cache architecture of *xeon*.

B Class Diagrams

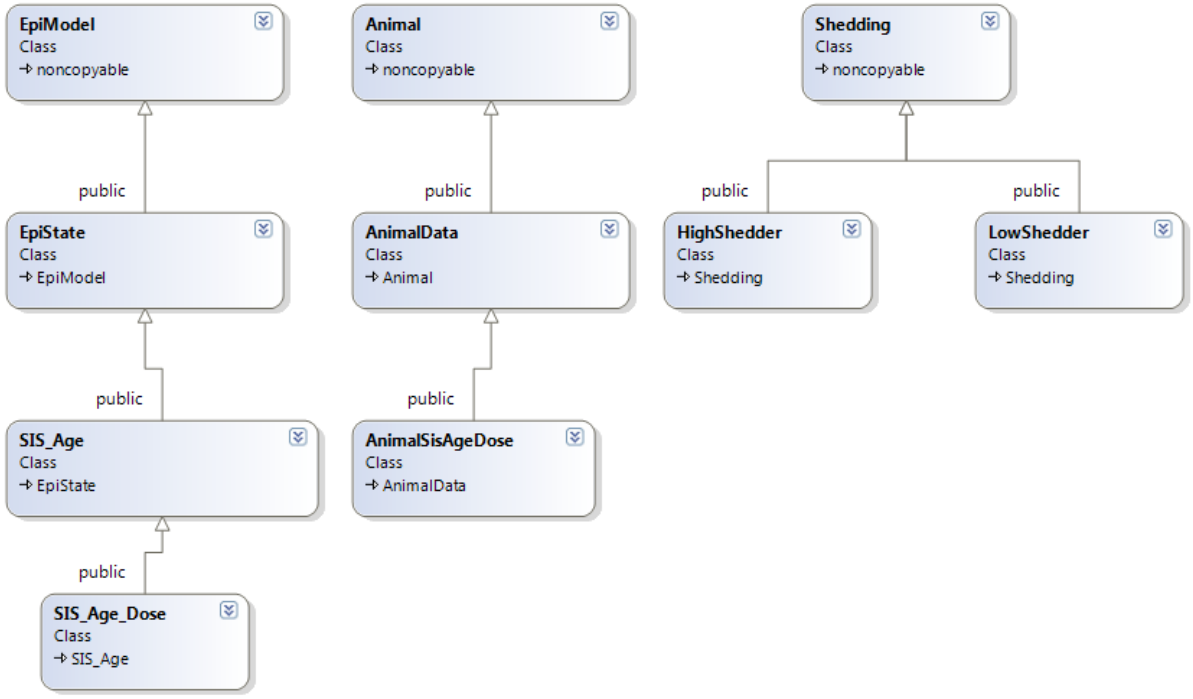


Fig. B-1: Class diagram of the original animal object.

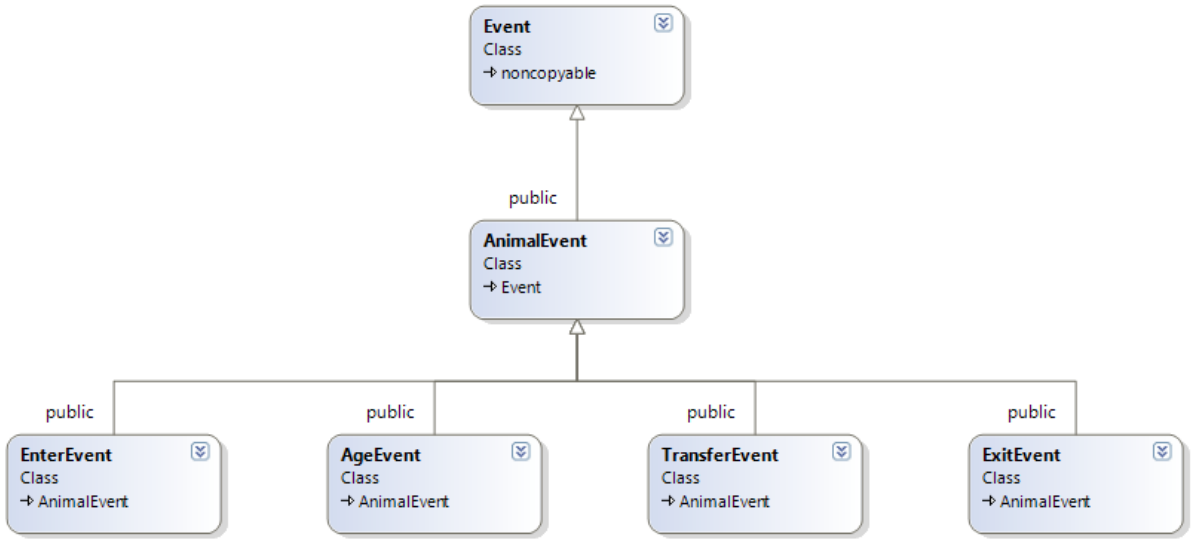


Fig. B-2: Class diagram of the original event objects.

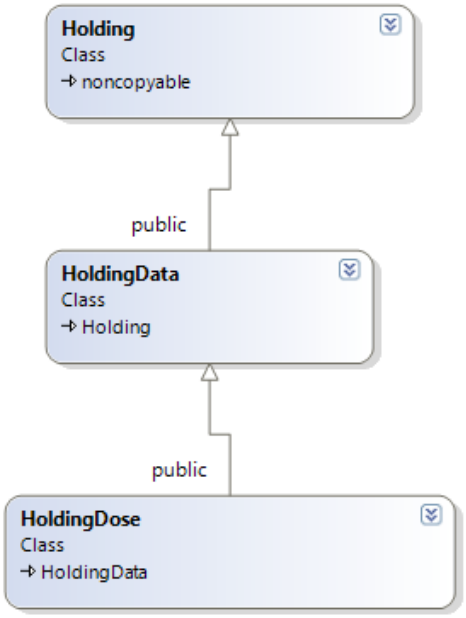


Fig. B-3: Class diagram of the original holding object.