An Individual Tree Simulator for Assessment of Forest Management Methods

Artur Signell[*] Johan Schöring[*] Mats Aspnäs[*] Jan Westerholm[*]

Abstract

Suswood is a parallel forest simulator capable of simulating growth, mortality and reproduction of large amounts of trees in multiple, polygon shaped compartments. Simulating on single tree level allows study of much more detailed management systems than on stand level, as tree selection can be made based on the properties of the trees, such as height and diameter or on the surroundings of a tree.

We describe the design, implementation, scalability and performance results of the simulator, which is fully parallelized and able to take advantage of the computational power of 1024+ computing nodes for a single simulation. It has been designed to be able to simulate forests larger than 100 000 ha containing billions of trees.

1 Introduction

Many forest simulators operate on stand level representing trees using tree distributions. In this approach average properties of the stands are obtained while local and individual tree properties are lost. In this paper simulations are done on a single tree level, simulating properties for every tree in the area. This approach, while requiring a lot of computational power, enables for instance the usage of different management methods where a decision to cut a tree is based on the individual properties of a tree. It is also possible to calculate the volume production more precisely: for any tree we can calculate the wood raw materials of different parts of the tree (stem, branches, stump, needles).

In order to get a more realistic simulation we have used multiple, arbitrarily shaped polygons as forest area shapes, contrary to the single and often quite small rectangle shape that is usually used in forest simulators, both on stand and individual tree level. Multiple polygons also enable the use of real forest data as a starting point for the simulation and to acquire simulation results which can be mapped directly onto real forests.

2 Individual tree simulator

The Suswood simulator is based on a statistical model[1]. The simulation area consists of one or more polygon-shaped areas which are referred to as compartments. Each compartment is defined by a set of border points and has its own parameters reflecting the soil, growing conditions etc. in the area. Figure 1 shows a sample area of 35 hectares used as a test case in this paper. This area represents a real forest area taken from Nurmes, Finland.

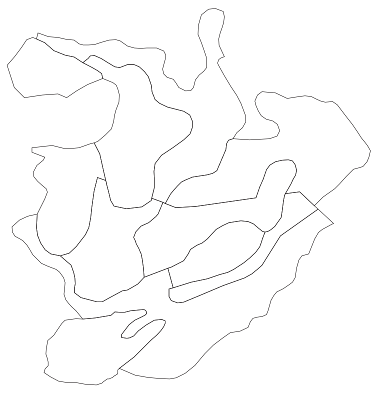

Figure 1: Sample area of 35 hectares taken from Nurmes, Finland

[*]Åbo Akademi University

2.1 Simulation flow

The simulation process is divided into two parts: simulation and statistics. The simulation can consist of one or more replications (repetitions of the simulation using different random numbers). Multiple replications are used especially in small simulations where one replication does not provide enough data to get statistically significant results. Each replication is divided into an initial generation phase and an iteration phase as illustrated in figure 2. These phases are described in more detail in the following sections.

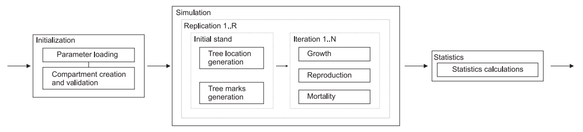

Figure 2: High level description of the program flow

2.2 Initial tree generation

The simulation can either be started from completely empty compartments, or initial tree positions can be loaded from a file. The empty compartments are populated using a Poisson point generator with given average and variance, which generates locations for the new trees. When the positions have been generated the model in eq. 1 is used to generate initial marks for the trees. A mark is essentially a degree of freedom from which tree properties can be derived, eg. tree height and trunk volume from the diameter mark. Currently the only mark used in the simulation is the tree diameter and consequently the model is here written in a form applicable to one mark and one tree.

$$Y = \underline{X}\underline{\beta} + \alpha\tau + \gamma\xi + \epsilon \tag{1}$$

In this model $\underline{\beta}$, α and γ are parameters given as input to the program. $\underline{X}$ contains the competition indices for the tree. The competition indices are the tree's diameter and some of the properties of the surrounding trees: density of the surrounding forest (number of neighbours / competition area), average distance to neighbouring trees, inverse average distance to neighbouring trees, mark sum of the surrounding trees and mark difference ($\sum$(target tree marks - neighbour tree marks)). When calculating the competition indices for a single tree all trees inside a given competition radius are taken into account. After competition indices for all trees have been calculated the means of the indices are calculated and are used to center the individual tree's competition indices around zero.

The second part of the model $\alpha\tau$ is a random component representing the initial heterogeneity of the tree. τ is generated using a random number generator when a tree is created and is stored so it will be the same for a tree during the whole simulation. α is a fixed simulation parameter denoting the weight of this component.

The third part ($\gamma\xi$) generates a correlated random effect for the tree. In order to calculate this we first have to generate a covariance matrix for the forest. The covariance matrix is generated using the covariance function (eq. 2). $|x_i - x_j|$ denotes the distance between tree i and tree j, θ_1 and θ_2 are input parameters and CR is the used competition radius.

$$C_{ij} = \begin{cases} e^{(\theta_1 + \theta_2(|x_i - x_j|))}, & \text{if } |x_i - x_j| \leq CR \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Correlated random numbers for the trees can be generated by using Cholesky decomposition[2]. By decomposing the matrix into a lower and an upper triangular matrix and multiplying the lower triangle part with a vector of uniform random numbers we get a vector consisting of correlated random numbers, one for each tree. The corresponding element from this vector is used as the γ parameter in the model.

The parameter $\underline{\epsilon}$ is a pure random component.

After calculating the competition indices and the correlated random number for each tree we can now calculate the initial marks (diameters) for all trees using eq. 1. The calculation of initial marks is done in one compartment at a time, independent of other compartments, using the parameters given for that compartment. Figure 3 shows the previous sample area after the initial generation phase. Each dot represents one tree.

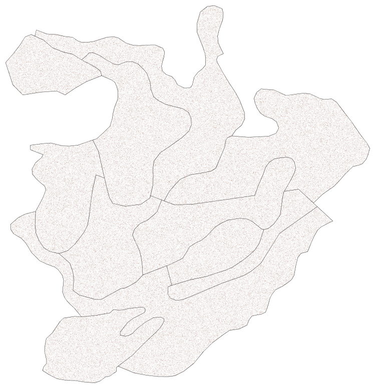

Figure 3: The sample area with 10 000 trees/hectare generated in the initial generation phase.

2.3 Simulation

After the initial generation phase is complete we have reached the starting point for the simulation. The actual simulation is done in iterations (time steps) with a typical length of 1-5 years. The simulation flow is divided into four parts: growth, reproduction, mortality and management. The order in which the parts are performed does not matter because every part operates on the tree data available at the beginning of the iteration. Updates for all parts are done at the end of the iteration.

2.3.1 Growth

Tree growth is calculated using the same model as the initial marks (eq. 1) but using different parameters. Competition indices are calculated by taking into account all neighbouring trees independent of which compartment they are located in. When centering the competition indices only the means for trees positioned in the same iteration are used. The covariance matrix is constructed and multiplied with generated, independent random numbers to acquire correlated random numbers in the same way as in the initial generation. Finally Y, which represents the tree growth, is calculated and stored so the tree diameters can be updated at the end of the iteration.

2.3.2 Reproduction

Reproduction is done in the same way as the initial generation. The number of new trees to be generated is calculated from the expected number of trees using the Poisson random number generator. New tree positions are then calculated using a chosen point generation process. Currently only a homogeneous point process is available in the program for generating the positions. The homogeneous point process generates an evenly distributed forest, which means that each position in the compartment is just as likely to be chosen for a new tree. The new positions are checked so they are not inside the trunk of an existing tree, but otherwise the placement is done exactly as in the initial generation. After the positions have been generated the competition indices and covariance matrix are calculated as in the initial generation phase, with the exception that only new trees are included in the covariance matrix. The new trees are then stored and are not included in the calculations until the next iteration.

2.3.3 Mortality

There are two models for mortality in the simulator. Firstly, if the tree growth is less than zero or than a mortality threshold, specified as a parameter, the tree will die. Secondly, the model (eq. 1) is used to calculate a mortality mark for each tree. The parameters used when calculating the mortality are mortality specific but the calculation is the same as in the other parts. The mortality mark is used as x in eq. 3 to calculate the probability p that the tree will die. If a random number which is drawn is less than p the tree will no longer exist after the current iteration.

$$p = \frac{1}{1 + e^{-x}} \tag{3}$$

2.3.4 Management

Management, which can be considered as a type of mortality, is performed as the last step of the iteration. A number of different management methods can be used in the simulation, eg. dimension cutting (removal based on tree diameter) or low thinning (removal based on tree height). The management system is still being developed and will therefore not be discussed further in this paper.

2.4 Statistics

The program supports outputting information (position and marks) for all trees that are alive at the end of the simulation, which is useful mainly for debugging purposes. For normal use we calculate summary statistics consisting of number of trees grouped by different factors such as size and iteration. We also calculate mean values for, among others, tree diameter, forest density and distance between trees. The total basal area of the trees in the forest is also calculated at the end of the simulation. In the near future the statistics will also contain volume calculations which will tell how many cubic meters would be acquired if a clear cut would be performed at the end of the simulation.

In addition to these statistics the program produces a frequency distribution table. Trees are divided into roughly 70 classes based on the their diameters, and the frequency (number of trees) for each class is output.

The last statistical output is a correlogram which contains the correlation between trees for predefined lags (distances from each other). Usually the correlogram is calculated for 10 different lags and the data should be decreasing by the lag number as the correlation between trees should get lower as the distance used for calculation increases.

All statistics are calculated and output separately for each compartment. If the simulation consists of several replications the statistics is calculated separately for each replication and the output consists of the minimum and maximum values from any replication and also the average for all replications.

3 Implementation

The SUSWOOD simulator is implemented in C++. When the project started it was decided to emphasize program design and structure, and postpone all possible issues with efficiency to a later stage in the program development. Using an object oriented program design approach has forced the developers to really understand the problem in order to create a good, clean class design which can easily be extended with new features.

3.1 Data structures

The main data structures in the simulation are used for representing trees and compartments. Each compartment consist of the associated polygon, which is described by a number of points, and a number of trees populating the compartment. Each tree must belong to precisely one compartment. Some parts of the program, such as reproduction, are done compartment by compartment and in these parts it is important to find all trees in a given compartment. Other parts, such as growth, are done on the whole simulation area at the same time. In these parts a central operation is to find the neighbours of the tree and calculate the result of the interaction between a tree and its neighbours.

For these cases to be implemented efficiently it must be possible to both find all trees in a compartment and also to find all trees near a certain tree, independent of which compartments they belong to. To be able to find all trees in a certain compartment each Compartment has a linked list of Tree objects, representing the trees in that compartment. The program always performs compartment based calculations on all trees so the only used linked list operations are iteration, insert and remove. These are all efficient and can be executed in linear time (constant time for insert and remove).

A TreeContainer class is used to keep track of all trees in the simulation. This class is built around a simple array of Tree objects. An array is of course inefficient for searching so an additional structure called RectangleContainer is used to support efficient searching.

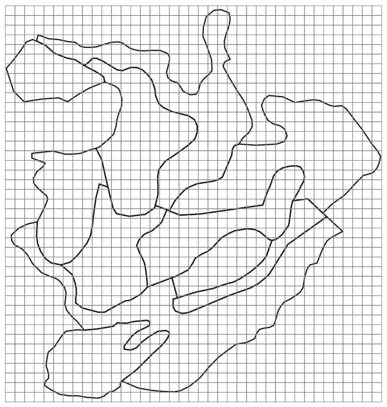

Figure 4: The sample area overlaid with 39x41 rectangles created by the RectangleContainer.

3.1.1 RectangleContainer

The RectangleContainer is a structure designed to support efficient searching for all trees within a certain distance from a specific tree. The RectangleContainer divides the area (the whole simulation area or a single compartment, depending on the situation) in x and y directions into many small, equally sized rectangles as illustrated in figure 4. After the rectangles have been created all trees in the area are added to the rectangle they reside in. If a tree lies within a specified distance from another rectangle it is also added to that rectangle. This means that a rectangle will contain all trees inside the rectangle area, and also the trees that are located within the specified distance from the borders of the rectangle. This is illustrated in figure 5.

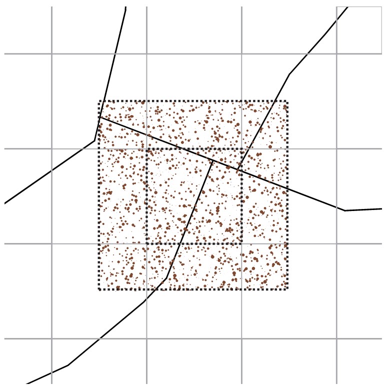

Figure 5: A rectangle contains the trees inside the rectangle and all trees within a given distance from the border of the rectangle.

To find the neighbours within a given distance from a certain tree we now fetch the rectangle where the tree is located. The rectangles are stored in a matrix, so it can be accessed directly. We then fetch all the trees belonging to the rectangle. Obviously this will also fetch a few extra trees that are not located within the given distance from the selected tree, so these have to be filtered out. This gives a list of trees that are close to the chosen tree. The list is traversed to find the actual neighbours inside the wanted distance from the tree.

The distance we are interested in may be different in different parts of the program. A distance smaller than the distance used when creating the rectangles pose no problem but if the distance is greater than the original distance the rectangles have to be discarded and reconstructed using the new distance.

3.2 Optimization methods

Certain well known code optimization methods have been applied to the program to improve its efficiency. Code inlining is critical as many small functions have been created to keep the code well structured. If full compiler inlining is not enabled, the program execution time may be up to twice as long as with full compiler inlining enabled.

The classes that consume the most memory have been carefully studied and all unnecessarily variables have been removed. The Tree class is the most instantiated class in the program and directly determines how large simulations can be run. The original version of the class required 80 bytes of memory and is shown in figure 6. The optimized version of the class, shown in figure 7, requires only 56 bytes per tree which is a 30% gain without sacrificing performance or precision of the final result.

Tree
Tree* prev,next Compartment* compartment double x,y double diameter double growth double heterogeneity unsigned int cohort, sizeClass bool dead

Figure 6: The original Tree class, requiring 80 bytes.

Tree
Tree* prev,next Compartment* compartment double x,y float diameter float growth float heterogeneity unsigned char cohort, sizeClass bool dead

Figure 7: The optimized Tree class, requiring only 56 bytes.

The life time of the Tree objects have also been studied carefully so that they exist in memory only when needed and are removed as soon as possible. This ensures that the maximum amount of memory is available for objects that are actually used and no memory is wasted on objects that are going to be deallocated in a later phase.

In addition to these class level memory optimizations we have also tried to reduce the dynamic memory allocations to a minimum. Instead of allocating a large number of small objects, memory is allocated in larger chunks and also freed in larger chunks.

For vector and matrix arithmetics the highly optimized ACML library [5] has been used wherever possible. Together all these basic optimizations has enabled simulations of notably larger areas in a clearly shorter time, compared to the original version of the simulator.

4 Parallelization

The sequential version of the simulator is mainly limited by the amount of memory available. A standard 2 GHz computer with 1 GB of memory can simulate an area containing roughly 4 million trees. To be able to simulate arbitrarily large forest areas, a parallel version of the simulator was developed.

The parallel version is based on a fairly standard domain decomposition where the simulation area is divided among the participating processes. Each process calculates how its own part of the forest evolves and occasionally communicates with the other processes. Since the simulated forest consists of arbitrarily shaped polygons which may have strongly varying tree densities, the decomposition has to be designed so that the processes get an even workload.

4.1 Domain decomposition

Two alternative domain decomposition were evaluated in SUSWOOD: assigning different compartments to different processes or dividing the area between the processes without taking compartment borders into account. As some parts of the simulation is done compartment by compartment and other parts are done independent of the compartment borders, it was not obvious which of these alternatives to choose.

Reproduction is done separately for each compartment: new trees are placed inside the compartment borders but trees both on the inside and outside of the borders affect the reproduction. If the decomposition would be done by assigning whole compartments to processes the processes would still need to be aware of the trees outside its compartments within the competition radius from the border. We will refer to these trees located in other processes as foreign trees, as opposed to own trees which are local to the process. As the compartments consist of arbitrarily shaped polygons, it is not very easy to identify the border zone which contains the foreign trees that other processes need to be aware of. For the growth and mortality phases also the information of foreign trees close to the own trees must be known, regardless if they are in the same compartment or not.

Based on these facts it was decided that the best way to divide the area between processes was to ignore compartment borders and do a decomposition where only globally straight lines separate an area belonging to one process from an area belonging to another process. This strategy has the advantage of giving very simple algorithms for locating trees to communicate to other processes: only trees in the border zone according to x or y coordinate need to be processed. A naive way of doing this decomposition would be to use the bounding box of the area and divide that into equally sized areas as shown in figure 8.

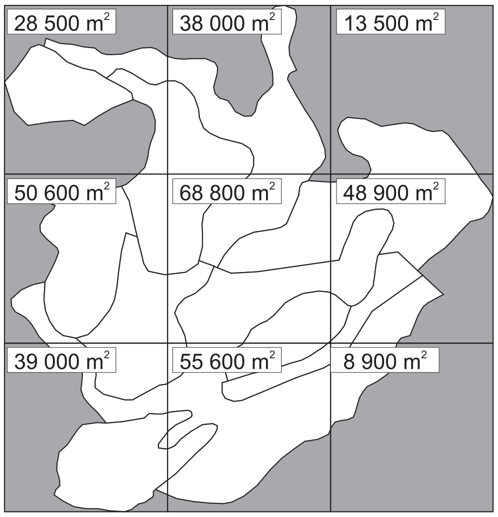

Figure 8: The sample area divided into nine sub areas for nine processes by using the bounding box.

A more sophisticated approach takes the workloads of the individual processes into account. The workload for each process is not directly related to the area that it is processing but rather to the number of trees that the process handles. The best way of dividing the area would thus be so that each process handles exactly the same number of trees. This is however not feasible, as we did not want to move the borders during the simulation. New trees are born and die during the simulation and change the density in the compartments, which would require borders to be moved for the equal load between processes to be kept.

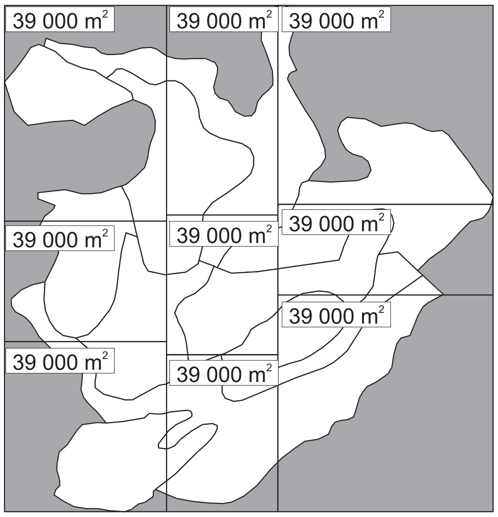

Figure 9: The sample area divided into nine sub areas so that each process is assigned an equally large area.

Because of this we settled on dividing the area based on the total forest area instead of number of trees. The decomposition of the sample area done this way is shown in figure 9. Even though there may be a clear cut of trees in a part of an area assigned to one process it is unlikely that the whole area assigned to the process would be clear cut, while other processes would contain areas with dense forests, making the workload unbalanced. On the other hand, even if this happened, new trees would grow in the clear cut area in the next iteration, balancing the workload once again. A process should thus not have a significantly different workload compared to the average workload during more than one iteration.

4.1.1 Decomposition implementation

The first step in the decomposition is to decide into how many rows and columns the area should be divided. The number of rows and columns are calculated so that the original aspect ratio of the forest area is taken into account and the generated cells are as close to squares as possible. The area of each process must be at least of size $2c$ x $2c$, where c is the competition radius, to avoid competition across more than two neighbouring subareas. This is best accomplished by dividing the full area taking the aspect ratio into account. If the acquired number of rows and columns do not use all available processes, another column is added at the right end of the area. All left over processes will divide this area among them, still in such a way that the area is equally large for each process.

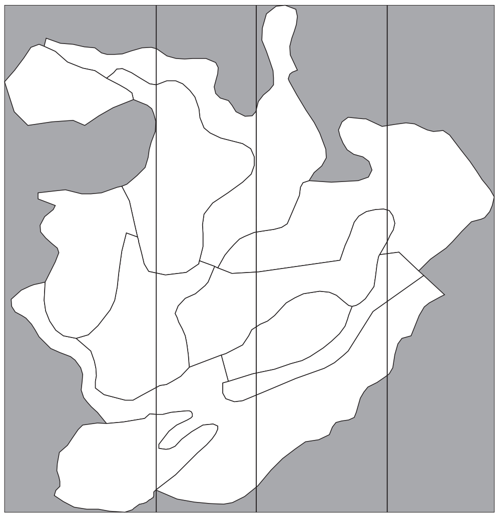

Figure 10: The sample area divided in x direction for 10 processes. The grid has been selected as 3x3 with an extra added column at the right end for the left over tenth process. The rightmost slice thus contains an active area of only 1/3 of the other slices.

When the number of rows and columns have been determined each process calculates the coordinates that limits its area. Based on the row number for the process (calculated from the process id) the process is able to determine how large fraction of the whole area should be to the left of its own area. In the same way the process can determine how large fraction of the area should be to the right. As the area is asymmetric it is not possible to directly determine where the borders should be based on the fractions, but instead a search has to be performed to find the actual border coordinates in x direction. Starting from the center of the area in x direction we split the area into two and calculate what fraction of the total area is to the left of the split coordinate. If this is larger than what we want, we split the left area into two and calculate the new fractions. This is basically a standard binary search algorithm which is repeated until the coordinate that gives the desired fraction is found, with a predefined precision. The actual value of the precision is not important as long as all processes use the same value and thus can agree on where the borders are. Small variations between the size of the area in different processes do not impact the overall performance. The result of this phase can be seen in figure 10 where the sample area has been divided in x direction when using 10 processes.

After calculating the boundaries in x direction we continue by calculating the border coordinates in y direction for each of the areas limited by the recently calculated x coordinates. The same algorithm is used for these calculations and as a result each process will know what area it is responsible for in the simulation. The final result of a decomposition of the sample area for 10 processes can be seen in figure 11.

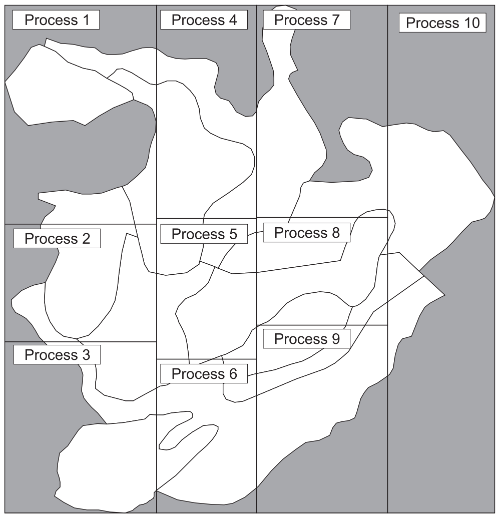

Figure 11: The sample area decomposed in both x and y direction for 10 processes. Each area is of equal size even though the last column contains only one row

4.1.2 Finding neighbouring processes

Each process must also be aware of which neighbours it has and which border(s) it has in common with its neighbours. In theory, this could be calculated from the information the process already has: the number of processes, its own position in the process grid and the locations of the borders. However, in order not to have to assume too much about how the decomposition is done the information about neighbours is communicated rather than calculated.

When each process has calculated the coordinates of its own forest area it broadcasts the corner coordinates (top-left and bottom-right) to all other processes. A process receiving this information can determine if the area is a neighbouring area or if it can ignore the information. For neighbouring areas it is also easy to determine which border the processes have in common.

4.2 Process communication

Since the simulator is designed to run on a Cray XT4[4] with a very efficient communication network, the design of the communication structure has not been aimed at minimizing the amount of data transfer. Instead, a more important goal has been to minimize the time the processes have to wait for each other, or in other words, to achieve a good load balancing.

In the initialization stage the processes communicate by broadcasting parameters, area coordinates and other initialization data, which takes a negligible amount of time. During the simulation communication is needed at the start of each iteration in order to exchange information about trees there are close to the border between two neighbour processes. The only communication that is needed during the simulation is in the reproduction phase, where processes that share some compartment exchange information about the competition indices of the new trees.

Reproduction is calculated in parallel for all compartments. All trees are placed inside the compartment area and the competition indices are calculated for all these trees. This computation takes roughly the same time in all processes, provided the simulated forest is homogeneous. When this is done for every compartment the processes exchange the average of the competition indices and can after that calculate the final marks for the new trees. The only drawback of calculating the reproduction in parallel is that the competition indices for every new tree have to be stored in memory at the same time, instead of only storing the competition indices for the new trees in one compartment. This is not a problem, however, as there are typically only 6 competition indices.

4.3 Computing statistics

Statistics on the trees in a compartment in the parallel version can be calculated in two different ways. For compartments that are not shared between two or more processes statistics is calculated exactly in the same way as in the sequential version. This is possible as the process has information about all the trees in the compartment.

For shared compartments, a single process does not have all the needed information and must communicate with other neighbour processes. Due to the nature of the statistical calculations it is not possible to calculate all statistics separately in each process and then simply combine the final results. The statistics is calculated in parallel, using the same methods as in reproduction. The memory consumption in this computation is, however, a slightly bigger problem as we need to calculate the competition indices for every tree owned by the process, as opposed to only the new trees in reproduction. The parallel statistics calculation is the heaviest part of the program, both for cpu load and memory consumption.

4.4 Validation of the parallel version

A problem encountered when developing the parallel version was how to validate that the parallel version performs the same calculations as the sequential version of the program. As the model is stochastic and uses random numbers for the simulation, it is impossible to directly compare the output consisting of tree locations and marks from the sequential and parallel versions. The results from two runs of the sequential version using different random number seeds will not be identical either. It is possible to compare the statistical summaries of the runs by hand to see that they are roughly the same. To be able to automatically compare the results from a sequential simulation with the results from a parallel simulation, a method for ensuring that both versions use the same random number sequences was introduced. This is called comparison seeding.

Comparison seeding is a special mode that can be enabled when the program is compiled. This mode introduces a seeding of the random number generator, which guarantees that both the sequential and parallel version of the code will use exactly the same sequence of pseudo-random numbers. In comparison seeding, the random number seeds used in the simulation are always based on something that can be derived from the current state of the simulation, and will thus be identical in both the sequential and parallel versions. Usually the seeding is done before an operation is performed on a tree and the seeds are based on a the position of a tree. Examples of this is calculation of initial marks and growth calculation. Reproduction, however, is a bit more complicated.

Reproduction is, as previously noted, performed on one compartment at a time. The polygon shaped area of the compartment is surrounded by its bounding box for the reproduction phase. A random position (x and y) inside the bounding box is generated for a new tree and the position is checked so it is inside the polygon borders and not inside another tree's trunk. In the parallel version the compartments can be shared between several processes, but the processes still have to use the same bounding box for the generated positions in order to be the same. Even when using the original bounding box the result will not be the same unless the process is aware of all trees inside the compartment and also all trees just outside the borders of the compartment, so the trunk comparison can be done correctly. This means that for comparison seeding to work in the parallel case the processes must send information about almost all their trees to their neighbours, significantly slowing down the simulation.

As the seeding in this mode is done based on the simulation state, the results of the simulation are predictable and comparison seeding should not be used for any production simulations. Comparison seeding is, however, a simple way to make the results from sequential and parallel runs directly comparable. Even though the seeding is based on properties of the simulation, it is still possible to vary the outcome by multiplying the seeds with different factors. Thus this does not limit the tests to a single case, but gives a possibility to do broader, more reliable test to show that the parallel version produces the same results as the sequential.

5 Performance

We have executed the program on the Cray XT4 at CSC[6] in order to measure the performance of the program and investigate its scalability. Figure 12 illustrates the scalability of the program. In these measurements the simulated area, and thus also the number of trees, is kept constant when the number of processes are increased so effectively the same simulation is executed using different number of processes. We see that the program scales very well (with a factor close to 2 when doubling the number of processes) up to 512 processes, and probably also well beyond this.

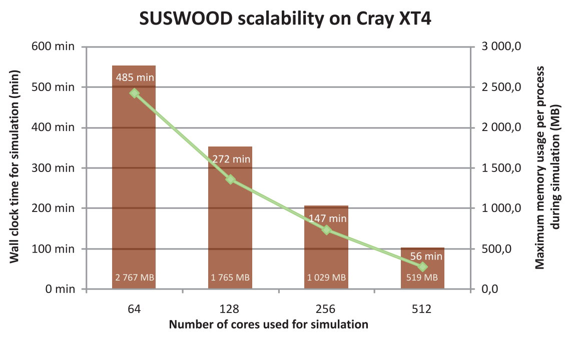

Figure 12: Runtime and memory usage for a fixed forest size when increasing the number of processes

6 Conclusions

This paper presents the design and implementation of a forest simulator where trees are represented as individual objects. The simulation model explicitly considers the location of trees and the competition between nearby trees to calculate the growth, mortality and reproduction processes in a forest area. The simulator is implemented as a highly efficient and scalable parallel program which can be used to simulate forest areas larger than 100 000 ha and containing over one billion trees.

Simulations of this kind will likely be a valuable instrument for large-scale forest management in the near future. Current technology already enables forest inventory by remote sensing, which gives detailed information about the individual trees growing in a forest, including position, species and size. This kind of data can be used as input to a forest simulator to investigate how the forest will evolve under different management methods, and to optimize the wood raw material production.

Acknowledgements

The facilities of CSC were used for this work. This work received support from the Academy of Finland in the KETJU[7] research programme.

About the Authors

Artur Signell is a Ph.D. student at the High Performance Computing laboratory at Åbo Akademi University. His research focuses on massively parallel computing problems. He can be reached at Åbo Akademi University, Department of Information Technologies, Joukahainengatan 3 – 5, 20500 Åbo, Finland, e-mail: artur.signell@abo.fi.

Johan Schöring is a research assistant at the department of information technologies at Åbo Akademi University. His work has been focused on optimization and parallelization of the object oriented forest simulator. He can be reached at joschori@abo.fi.

Mats Aspnäs has a Ph.D. in computer science from Åbo Akademi and works as a lecturer in parallel computing. He is currently involved in several software development projects for large scale high-performance systems, in cooperation with CSC. He can be reached at mats@abo.fi.

Jan Westerholm holds a chair on high performance computing at Åbo Akademi University in Finland. His main interests include code optimisation, parallel programming and non-linear optimisation applied to physical problems. Jan can be reached at jan.westerholm@abo.fi.

References

[1] Chijien Lin, Generating Forest Stands with Spatio-Temporal Dependencies Publications in Social Sciences 64. University of Joensuu 2003. 123 p.

[2] Numerical Recipes: The Art of Scientific Computing, Third Edition William H. Press et al. Cambridge University Press, 2007

[3] William Gropp and Ewing Lusk and Anthony Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface MIT Press 1994

[4] Cray Inc., The Supercomputer Company `http://www.cray.com/products/xt4`

[5] AMD Core Math Library. `http://developer.amd.com/tools/acml`

[6] Finnish IT Center for Science (CSC) `http://www.csc.fi`

[7] Sustainable production and products research programme KETJU `http://www.aka.fi/ketju`