COSC3500 Ant Colony Simulator Project

Milestone 2

Matthew Young
m.young2@uqconnect.edu.au
46972495

November 2022

Abstract

Agent-based models and biologically inspired computing are two popular methodologies to address many problems in computer science. However, despite prior work, optimising these types of algorithms remains a challenge. In this paper, I present a high-performance, biologically-inspired ant simulator as my project for COSC3500 High Performance Computing. I contrast its performance with three different implementations: serial, OpenMP and Message Passing Interface (MPI). A practical deployment on UQ's getafix HPC cluster is demonstrated.

Contents

1 Introduction

1.1 Research background

Agent-based models are a popular simulation technique in computer science. Such models have been applied for more than 20 years to solve various different problems, from business and economics to disease outbreak modelling [1]. Agent-based models simulate individual entities, called agents, and their individual decisions and their interactions with the environment. From these relatively simple rules, complex behaviour emerges.

Bio-inspired computing is a related technique in computer science that seeks to solve problems using techniques inspired by the behaviour of organisms in nature [2]. Many optimisation problems that are being solved nowadays have large state spaces and noisy input variables. These factors mean that traditional, mathematical optimisation algorithms can fail to converge, or take a very long time to do so. Recognising these factors, researchers have developed biologically inspired *meta-heuristics* for use in optimisation. Meta-heuristics are a broad class of procedures for solving numerical optimisation problems that doesn't guarantee convergence, but nevertheless typically has good results in large, noisy problems [3]. Many meta-heuristics use the nature of *swarm intelligence* from bio-inspired computing: the idea that many simple individuals can combine together to produce an overall intelligent system.

A popular bio-inspired meta-heuristic is ant colony optimisation. This algorithm is designed to solve optimal graph routing problems, and works by simulating virtual ants following a pheromone trail [4]. Ants initially wander the space randomly, leaving pheromone trails back to their colony. Once they find a potential solution, they leave pheromone trails towards the solution so that other ants can follow. Over time, the ants prefer to follow pheromone trails rather than wandering randomly, reinforcing existing trails. However, the pheromone trails also begin to evaporate over time as well. Therefore, slow paths gradually evaporate over time, while fast paths are reinforced. All of these factors combine together to produce a very robust algorithm that can converge on a solution quickly, optimise it over time (even subject to a changing environment), and avoid undesirable local minima.

Ant colony optimisation algorithms see widespread use in complex routing problems such as the travelling salesman problem [5] [6]. Previous literature shows the ability to run on high-performance computing clusters, including on GPUs [7]. However, as will be shown later in the report, ant simulators are still very complex and exhibit cache-unfriendly behaviour, making their effective optimisation a challenge.

1.2 Simulation design

Taking inspiration from the concept of ant colony optimisation, and bio-inspired computing, the real-world behaviour ants was investigated to develop the rules for the simulation.

The rules of the simulation are summarised in the following list:

- The simulation operates on discrete integer time steps.
- The world is a 2D discrete grid consisting of food tiles, pheromone tiles, obstacle tiles and empty tiles by default, specified by the loaded map file.
- Each colony occupies a square grid of tiles and starts with a certain number of base ants.
- Ants can only occupy one tile in the grid, and can move in the cardinal directions (e.g. north, north east, etc). Moving diagonally costs the same as moving forwards, so ants use Chebyshev distance.
- When ants move, they lay down pheromone trails that indicate the direction to food or to their colony.
- Pheromone trails evaporate by a certain amount every tick.
- Ants initially "burst out" from their nest in a circular pattern and wander randomly until they reach food.
- When they reach food, ants turn around and return to their colony.
- When an ant returns to a colony, the colony's hunger is replenished by a certain amount. Each tick when an ant has *not* delivered food to a colony, its hunger level is diminished by a certain amount.
- When an ant colony has full hunger, it is allowed to spawn a certain number more ants.

- If an ant has not touched food or delivered food in a certain number of ticks, it is considered "useless" and is killed.
- The simulation ends when all food has been eaten, all colonies have died, or the config parameter `simulate_ticks` has elapsed - whichever happens sooner.

The above rules can also be represented as pseudocode:

```
function updateAnt(ant) is
    let newX = ant.x, newY = ant.y
    let pheromoneStrength, pheromoneVector = computePheromoneVector(ant)

    if pheromoneStrength >= antUsePheromoneThreshold:
        // pheromone strength is high enough, use it
        let movement = phVector
    else
        // pheromone strength is too low, move randomly
        let movement = randomMovementVector()
    newX, newY += movement

    if (ant not in bounds or ant intersects obstacle or (ant holding food and ant on food)):
        // reached an obstacle, flip the preferred dir
        ant.preferredDir *= -1
    else
        // checks passed so update ant data
        ant.pos = newX, newY
        // record where the ant walked, so it doesn't go back and forth
        insert movement into ant.visitedPos

    if ant holding food:
        // holding food, add to the "to food" strength, so we let other ants know where we
        // found food
        increment pheromoneGrid(x,y) for ant colony and state "to food"
    else:
        // looking for food, update the "to colony" strength, so other ants know how to get home
        increment pheromoneGrid(x,y) for ant colony and state "to colony"

    // update ant state
    if ant holding food:
        ant.holdingFood = true
        clear ant.visitedPos
        ant.preferredDir *= -1
        ant.ticksSinceLastUseful = 0
    else if ant is holding food and ant is close enough to its home colony:
        clear ant.visitedPos
        ant.preferredDir *= -1
        ant.ticksSinceLastUseful = 0
        mark ant colony for adding more ants/boosting (after loop)

    if ant is not holding food:
        ant.ticksSinceLastUseful++

    // kill the ant if it has not been useful in a while
    if ant.ticksSinceLastUseful >= antKillNotUseful:
        ant.isDead = true
```

2 Serial implementation

Before attempting optimisation strategies like intrinsics and multi-threading, a single-threaded (serial) implementation was developed as a baseline.

2.1 Program design

2.1.1 Toolchain

The language used to develop the simulator is C++17. C++ was chosen specifically because it's easy to optimise and designed for high performance computing.

CMake was used as the build tool, and the project was designed to compile equally as well under recent versions of both Clang and GCC. Clang 15 was used locally for development, and GCC 10 was used on the getafix HPC cluster.

2.1.2 Grid data structure

Arguably the most important aspect of any simulation is the data structure used to represent it. This can drastically impact the future possible total performance of the simulator. A difficult choice was encountered between the concept of a discrete and continuous grid data structure, as each have their own trade-offs and benefits. Continuous spaces should be easier to vectorize, but discrete spaces have much better cache efficiency. In the end, a discrete state space was chosen. Most existing ant simulators use a discrete state space, it seemed to make the most sense considering how pheromones are represented. As it turns out, there are many opportunities to vectorise the current code, so it should be a non-issue.

2.1.3 Simulator initialisation

The simulator has many constants that may need to be tuned at runtime, preferably without recompiling. Often times, `#define`s are used to configure simulation parameters, but for this simulator, an INI file and parser was added that an end-user can edit to change most of the simulator's parameters each time it's run.

The initial grid layout is loaded from a PNG file. The width and height of the PNG become the dimensions of the simulation world. Green colour (RGB 0,255,0) represent food, black RGB (0,0,0) represents empty cells, grey (RGB 128,128,128) represents an obstacle the ant cannot cross over, and any other colour of one pixel is considered the spawn point for an ant colony. This approach allows easy and fast creation of new maps to experiment with, and saves programming work instead of creating a custom tool.

2.1.4 Serialisation

In order to record and play back the simulation for analysis and verification, a serialisation format was designed, with the aim to be easy to program, lightweight (in terms of file size) and fast to encode.

A format called "PNG TAR" was derived, which is simply a sequence of PNG files in an uncompressed tarball (plus a small amount of simulation metadata). The discrete nature of the grid allows easy encoding into image files, and the losslessly compressed PNG was a natural choice so as not to lose any quality. Once written, the PNG TAR archive can be downloaded off the HPC server and played back using a simple Python script. While more efficient formats are possible (e.g. differential encoding between frames would save a lot of space), I believe this is a good approach given the relevant constraints.

The following image (Figure 1) shows an example of PNG TAR output:

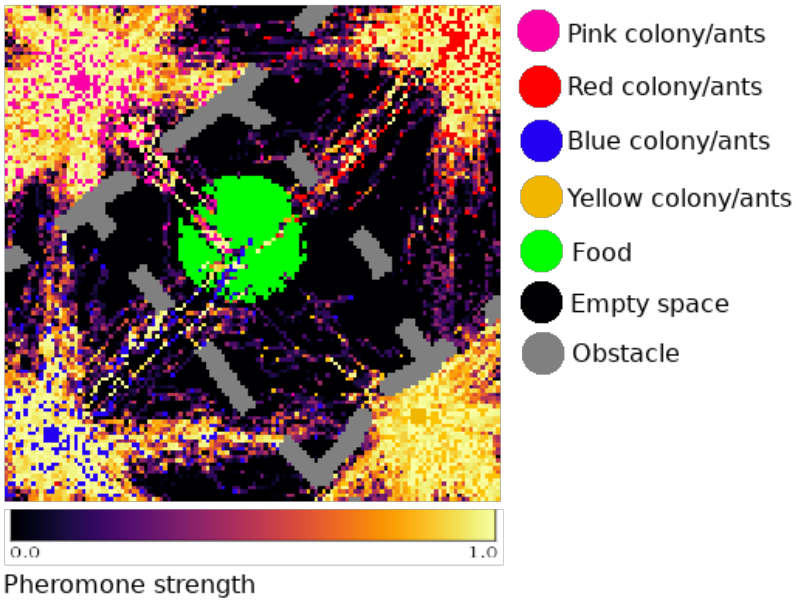

Figure 1: A screenshot from a PNG TAR recording, with a legend explaining the colours.

2.1.5 Random number generation

Random number generation is an important part of most simulations, including this one. Both the `rand()` function built into the C standard library, and the Mersenne Twister in the C++11 standard library, are widely used, but recent research has suggested they can be problematic [8].

More recent research into pseudo-random number generation has yielded smaller, faster algorithms with significantly better quality. One highly regarded algorithm is PCG [9], which is very fast and has excellent statistical quality, measurably better than `rand()` and measurably faster than the Mersenne Twister. A single, shared RNG instance is used throughout the whole simulation, but the large state-size of PCG makes this acceptable. The PCG instance the simulator uses is seeded with a configurable fixed seed so that outputs can reasonably be compared.

In the ant colony simulator, randomness is very important. When ants are moving initially before any pheromone trails exist, they move randomly to try and locate food. Randomness was also observed to significantly improve the results of the simulator when decaying old pheromone trails. Adding a small amount of uniform random noise to this threshold improves the rates of ants finding food significantly.

Given that one of the key aspects for the verification of this simulator is to keep it 100% deterministic between runs, a random "seed" is used to initialise PCG to the same random sequence each time. This way, it's still high quality, but repeatable and does not impact determinism.

2.2 Compiler flags and optimisation

CMake allows for multiple release targets, so "Debug", "Release" and "Profile" targets were implemented.

The debug build enables the Address Sanitizer and Undefined Behaviour Sanitizer tools [10], to detect any potential memory corruption or undefined behaviour. The binary is also compiled with `-O0`, i.e. without any compiler optimisations enabled at all, to ease debugging.

In the Release target, more optimisations are enabled to achieve higher performance. The optimiser level was set to `-O3`, the highest safe level available (`-Ofast` is typically considered unsafe for numerical work [11]). In addition, the flags `-march=native -mtune=native` were set, which generates the most optimal code for the CPU architecture the code is being compiled on (e.g. enabling the auto-vectorizer or micro-architecture specific optimisations). Finally, link-time optimisation, which is a new whole-program analysis technique, was also enabled.

Another target for profiling was also created. Optimisation is disabled in this target because the compiler's aggressive inlining makes the results difficult to interpret, and compiler optimisations may "hide" underperforming code patterns.

2.3 Verification

The nature of this type of simulation means there's no absolute reference to compare the output to and show it's 100% accurate. However, a few criteria can be looked at to see if the simulation is working correctly:

Do the ants generate and follow an optimal route to the food?

Several ant colonies did actually exhibit the behaviour of generating an optimal route to the food. Unfortunately, the simulation parameters are very sensitive so tuning them will take quite some time. As mentioned in the random number generation section, adding random noise to some of the simulation parameters improves the number of ants finding food significantly, and so a few parameters like the number of ticks after "useless" ants are killed and the pheromone decay rate, are fuzzed with small amounts of uniform random noise.

Are all the simulation mechanics (detailed in section 1.2) behaving correctly?

The simulation mechanics should be behaving correctly now. There were some issues that were discovered in regards to clamping the pheromone values and colony hunger, but they have been fixed. The general algorithm of the simulator is very similar to the well-documented ant-colony optimisation algorithm. If any issues remain, it's a matter of parameter tuning and minor fixes.

Does the simulator crash or enter into undefined behaviour?

The simulator doesn't crash or enter into undefined behaviour in normal cases. The Address Sanitizer was run over the program many times, and the output logs (see Appendix C) were scrutinised, but no issues were discovered. It has been observed that excessively large grid sizes with huge numbers of colonies (generally above 1000) cause memory exhaustion (they try to allocate over 200GB of RAM), so it is advised to keep the number of colonies under 1000.

Is the simulator deterministic between different runs?

To accurately compare performance characteristics of this particular simulator, it's very important it is deterministic between different runs (i.e. the output is always the same, given the same seed). To ensure this during development, a script was written to take the SHA256 checksum of all PNG files in the PNG TAR recording archive. If the checksums do not match, then the code has broken the simulation. This also, in turn, proves that the simulator is deterministic between runs.

2.4 Performance results

Performance of the simulation was measured using a few metrics: wall time (milliseconds), simulation time (milliseconds), wall ticks per second (TPS) and simulation ticks per second (TPS). The difference between "wall" and "simulation" time is that "wall time" includes the time spent writing the PNG TAR and initialising the simulator, whereas simulation time is just time spent updating the world. Since the goal of this project is to optimise the simulator, the metric used in this and the next section will be simulation time, not wall time.

The following benchmarks (Figure 2, Figure 3, Figure 4) were performed on my personal workstation, which has a 32-core, 16-thread AMD Ryzen 9 5950X CPU and 32GB of DDR4-2400 RAM. The simulation was run without recording enabled.

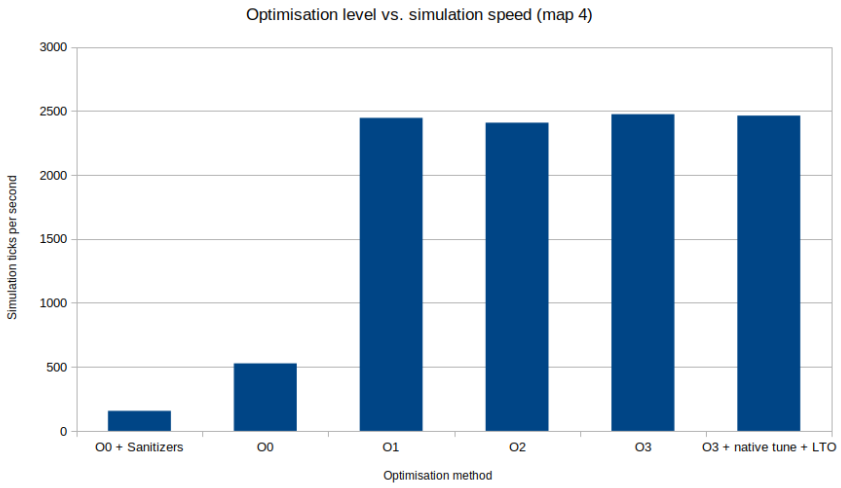

Figure 2: Performance of the simulator using different LLVM optimisation levels, on an AMD Ryzen 9 5950X CPU

The "O0 + Sanitizers" column (first one) is the Debug build. The "O3 + native tune + LTO" (last one) is equivalent to the Release build. The graph shows that changing between O0 and O1 yields the most significant

increase in the program's performance. From there on, O1 to the CMake Release target all have the same performance. This is somewhat surprising, and a little dissappointing, but compiler optimisations can't be guaranteed to speed up every piece of code ever.

It's worth nothing that the results in this version of the report (milestone 2) are very different to how they were in milestone 1. In milestone 1, the biggest difference was between O1 and O2. The grid data structure was changed between these two revisions (see section 3.1.1), and the ant colony update loop was refactored to perform more serial colony update work at the very end. This is probably the reason the compiler optimisations perform differently. Nonetheless, it's still very interesting that two versions of the same algorithm can produce vastly different performance results.

It was hypothesised that grid size impacts simulation performance. To test this, map 4 was resized from its original 128x128, to 256x256, 512x512, 1024x1024 and 2048x2048. The Release build was tested on each map, yielding the following graph:

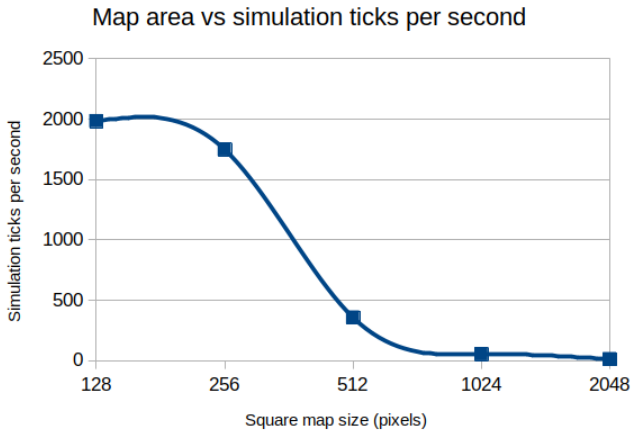

Figure 3: Map area vs simulation ticks per second. The map is square, and its size is on the x-axis.

As can be seen, the simulation becomes exponentially worse in speed as the map area increases. Parallelising the update function should help to alleviate this performance issue, which will be attempted in future.

It was also hypothesised that the number of ants in the simulator affects its performance. To test this, the average iteration time in milliseconds for the simulator was recorded as a function of the number of ants in the simulator, and plotted to produce the following graph:

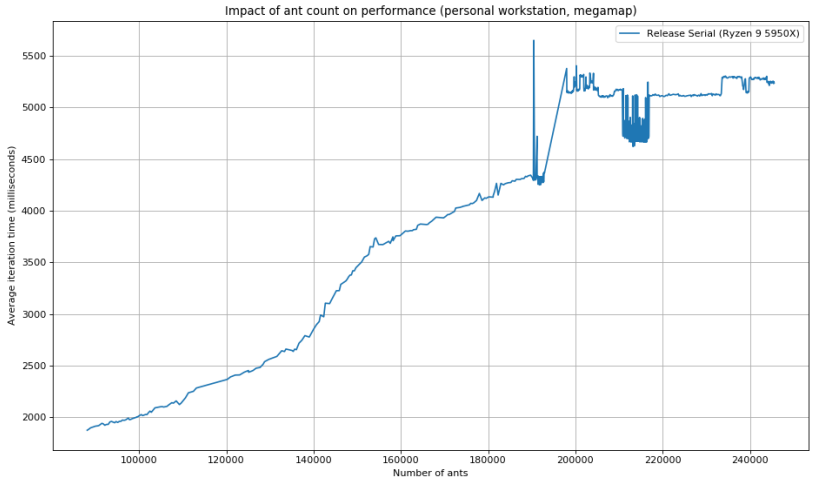

Figure 4: Impact of ant count on performance, on an AMD Ryzen 9 5950X CPU

This graph has an interesting trend. Simulation slowdown appears to scale approximately exponentially

(or possibly linearly, depending on what trend you want to fit) up until about 200,000 ants. Then, there is a sudden jump, the graph becomes very noisy and the general trend appears to be that the simulation doesn't slow down any more. It's not exactly clear what causes this jump, but it may have to do with certain compiler optimisations or cache boundaries.

2.5 Profiling

Using the Profile target in CMake, Valgrind's Callgrind and Cachegrind tools were used to profile the simulator and its cache performance, as well as to determine code hot spots for optimisation. These tools have very accurate (cycle accurate) logging, at the cost of extreme program slowdown, but their results are much more accurate than instrumentation-based profilers like gprof.

PNG TAR recording was disabled so the profiler would focus only on recording code, and the simulation was run through Callgrind and Cachegrind for 1500 ticks using map4 (128x128 with 4 colonies). The reports generated by the Valgrind tools were then loaded into KCachegrind, a GUI application for analysing Cachegrind and Callgrind results.

The results were ordered by the "Cycle Estimation" cost type, which estimates how many CPU cycles a function takes, to determine the most expensive C++ functions. The top relevant results were:

1. `ants::World::update` (98.67%)
2. `ants::World::updateAnt` (60.26%)
3. `ants::World::computePheromoneVector` (46.56%)
4. `std::set<>::find(ants::Vector2i)` (39.81%)
5. `ants::World::decayPheromones` (33.82%)

As might be expected, these results indicate that the world update function takes up almost all the time in the simulator (especially since PNG TAR recording was disabled). Given that `updateAnt` only includes code to update each ant, not the remaining colony tasks (e.g. spawning more ants, updating colony alive/dead, etc), we can conclude that the ant update loop is a particular hot spot that is a good target for code optimisation. Also interesting is how expensive `computePheromoneVector` is, given how relatively simple the function is. It is also worth noting how expensive finding each visited position for each ant in a set is (`std::set<>::find(Vector2i)`). Although a set data structure was used for fast lookups, it still may not be fast enough, or a better performing implementation should be used instead. Finally, it's interesting to note that `decayPheromones` is *relatively* cheap given that it iterates over the entire grid - this may be a product of the small grid size used, and not indicative of overall performance characteristics.

A call graph, commonly used in profiling analysis, was also rendered (see Figure 5).

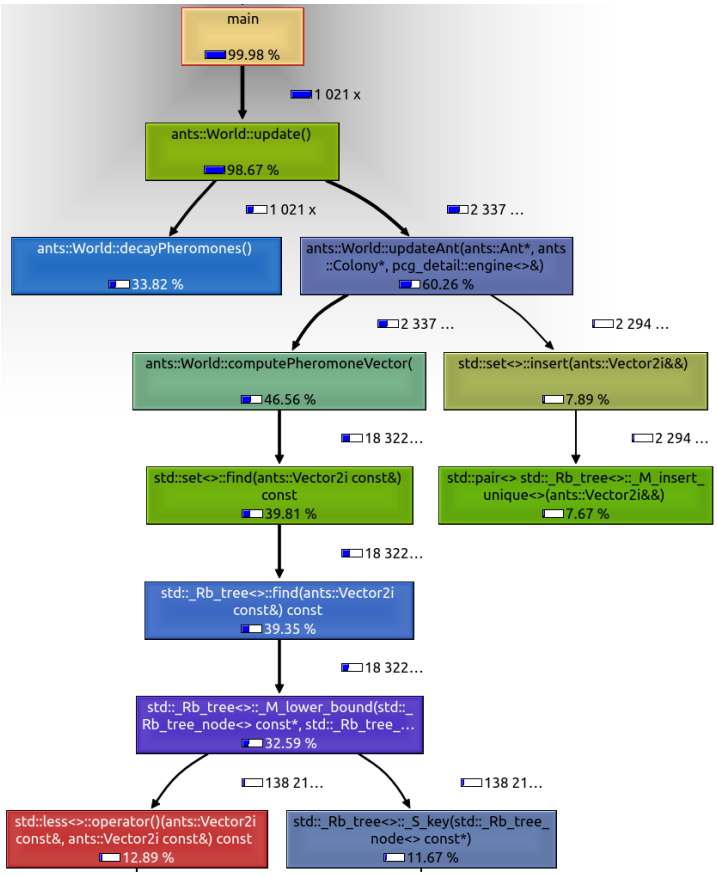

Figure 5: Callgrind profiling as rendered by KCachegrind. The cost type is set to "Cycle Estimation".

Using KCachegrind and Valgrind's Cachegrind tool, the functions with the most cache misses were also determined (see Figure 6). In modern processors, cache efficiency plays a vital role in program performance.

Incl.	Self	Called	Function	Location
99.96	0.01	1	main	ant_colony
98.21	3.75	1 021	ants::World::update()	ant_colony
41.70	7.35	1 021	ants::World::decayPheromones()	ant_colony
38.77	0.25	2 337 243	ants::World::updateAnt(ants::Ant*, a…	ant_colony
35.27	0.00	2 337 243	ants::World::computePheromoneVe…	ant_colony
27.07	27.07	32 681	__memcpy_avx_unaligned_erms	libc-2.31.so: memmo
26.65	0.00	2 042	ants::SnapGrid3D<>::commit()	ant_colony
26.38	0.00	18 322 606	std::_Rb_tree<>::find(ants::Vector2i …	ant_colony
26.38	0.00	18 322 606	std::set<>::find(ants::Vector2i const…	ant_colony
25.92	0.00	18 322 606	std::_Rb_tree<>::_M_lower_bound(s…	ant_colony
21.73	21.73	175 738 714	ants::Vector2i::operator<	ant_colony
21.73	0.00	175 738 714	std::less<>::operator()(ants::Vector2…	ant_colony
21.03	21.03	66 912 256	ants::PheromoneStrength ants::Sna…	ant_colony

Figure 6: Listing of functions by most L1 cache read misses, KCachegrind

The reason for the slow speed of `decayPheromones` is immediately obviously: it has the most number of cache misses out of all the functions developed for this simulator, by a high margin. This is most likely because the world data structure is simply to large to fit in most processor's L1 caches (which are only a few Kb), and also amplified by the fact that the arrays are probably not stored in a contiguous block of aligned memory.

Another interesting find is that the SnapGrid3D (covered more later in the report), referring to the pheromone grid, has a high cache miss rate. This is also related to the `__memcpy_avx_unaligned` function call, which proves that unaligned memory is being used and weakening cache performance.

One other puzzling result is how apparently expensive the `ants::Vector2i` less than operator is. This is quite confusing because it just compares two members of a struct (`x < other.x && y < other.y`), but may be due to bad cache utilisation patterns.

Overall, the use of Callgrind and Cachegrind has greatly helped to find code hot spots for optimisation. The fact that `updateAnt` is so expensive is somewhat of a good result, in fact, because it goes to show that the

simulator can be optimised significantly by multi-threading.

2.6 Conclusion

So far, the serial implementation of this biologically inspired ant simulator has been written so far. It has been written in C++17 and includes features like serialisation to the "PNG TAR" file format. Many different factors, from serialisation to highly efficient pseudo-random number generation, have been taken into account to produce an accurate and fast simulator. Hot spots in the code have been identified, and performance results have been compared between different compiler optimisation levels. Areas for parallelising the code have been suggested.

3 Parallel implementation

The key to improving the performance of this simulation on HPC clusters should be to use parallel computing. This comes in many forms, but for the ant colony simulator, multi-threading via OpenMP and multi-node acceleration via MPI will be the methods used.

3.1 Methodology

The key to speeding up this particular simulator is parallelising the ant update loop, which as the profiling earlier showed, takes the majority of the time. While other techniques like improving the cache-efficiency of the grid data structure or use SIMD would help, the most dramatic performance improvements for HPC clusters specifically will involve combining the massive parallelism of these clusters with the natural potential parallelism of the problem.

Typically, paralllelising simulators like this one involve subdividing the grid and simulating each subdivision independently [12]. This is typically done because, for example in a particle simulator, particles have dependencies on each other. However, in the ant simulator, there are no inter-dependencies between each ant. The ants don't communicate or interact with each other, only the pheromones around it and its parent colony. In other words, the ant state update function is a pure function, as long as the world state remains constant during the loop.

The above realisation is the foundation of optimising this particular simulator. By creating a new data structure to exploit the nature of independent ant updates, we can simply run each colony or each ant on its own processor/thread, and almost immediately reap the benefits of massive parallelism.

3.1.1 SnapGrid: Parallel grid data structure

In order to parallelise these ant updates, a new data structure called the "snapshot grid" or SnapGrid was invented. The SnapGrid maintains two buffers: the clean buffer, and the dirty buffer. It has three functions: read, write and commit. During the ant update loop, when the SnapGrid read function is called, it always returns data from the clean buffer. When the write function is called, the writes are redirected to the dirty buffer. Finally, after the parallel loop is complete, the commit function can be called, which copies the updated dirty buffer into the clean buffer. This is further explained in Figure 7.

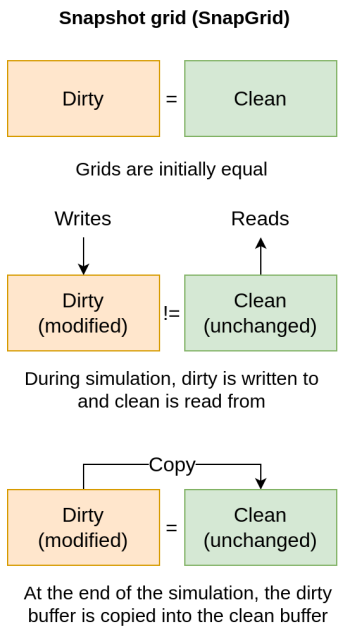

Figure 7: Diagram explaining the snapshot grid (SnapGrid) lifecycle

SnapGrid is the real fundamental breakthrough to parallelise this simulator. The ant update function is pure on a per-ant basis, which means that all ant updates can be computed in parallel, as long as the world remains the same - which SnapGrid was created to ensure. In the parallel code, SnapGrid is the primary synchronisation primitive, along with OpenMP critical sections for common read-write patterns.

The implementation of SnapGrid in the simulator makes use of C++ templates. There are two classes, `SnapGrid2D` and `SnapGrid3D`. The 2D version has (x,y) indexing, whereas the 3D version has (x,y,z) indexing.

In both implementations, the data is stored as a contiguous array of the template datatype `T`, for easy copying with MPI.

One disadvantage of the SnapGrid, with its clean and dirty buffer, is higher memory usage. This is especially noticeable on larger grid sizes, and because of this, unfortunately the ant simulator is extremely memory hungry. It has been known to use almost 30 GB of RAM to render a 960x540 map, and rendering a 1280x720 map is impossible because it would require more than 200 GB of RAM.

3.2 OpenMP

Adapting the existing serial code to use multi-core processors is an easy yet effective place to start HPC improvements on. Arguably the easiest and most versatile tool to achieve this goal is OpenMP (OMP). OpenMP is a set of C/C++ directives (e.g. `#pragma omp parallel`) combined with a runtime library that allows easy parallelisation of common code patterns like for loops. However, OpenMP does not ensure synchronisation between threads, so this must be taken care of by the programmer.

To adapt the ant simulator to OpenMP, the serial implementation was first rewritten to use the new SnapGrid data structure. Then, the colony update loop was surrounded in an OMP parallel for block. Typically, OpenMP is used to parallelise the outer loop (not the inner loop) - hence why it was applied to the colony loop, not the ant update loop. Sections involving writes to the grid, or read-writes, were surrounded in OMP critical section blocks to try and prevent race conditions. As will be covered in the verification section, this doesn't fully ensure 100% determinism in the simulator, but is a good first step.

Additionally, the body of the `decayPheromones` function was also multi-threaded in an OMP parallel for block in much the same way as the colony loop, since it had been previously identified as a code hot spot in the serial section.

Not all of the simulator can be parallelised in OpenMP. There are still a number of "serial tasks" that are performed at the end of the colony loop that are not parallelised. This includes spawning in new ants when a colony is at full hunger, updating colony stats, committing SnapGrids and counting the amount of food remaining. These tasks have specifically been chosen to run serially so as not to break determinism or cause race conditions.

Although deterministic random number generation is very important in this simulator, unfortunately the implementation of the PCG RNG used is not thread-safe. When multi-threading, depending on the Linux thread scheduler, each thread could get an unpredictable view of the random sequence, which would break determinism. To fix this issue, each thread creates its own "thread-local" `pcg32_fast` instance, which is seeded with a random `uint64_t` generated by the master thread.

Overall, the OpenMP implementation was relatively easy to achieve and shows significant improvement over the serial version as grid size increases.

3.3 MPI

Message Passing Interface (MPI) is a parallel computing framework very commonly used in the world of high-performance computing. Paired with a high-speed interconnect, it allows massively parallel execution of programs across many different physical machines. Whereas OpenMP allows parallelisation within a single compute node, MPI allows parallelisation across an entire datacentre.

While OpenMP uses shared memory between processes, so the code can mostly stay the same, MPI requires a total re-design of the simulator. Not only will the program's code need to be divided between the master process and its MPI workers, an efficient method for transmitting and receiving the simulation state essentially over network will need to be devised.

The following pseudocode was designed to implement MPI:

```
// main.cpp
Master/Worker: Run init code. All workers now have same state.

// World::update
Master (Rank 0): Broadcast the clean buffer for each SnapGrid to all workers.
Worker (Rank N): Receive the clean buffer and copy into the diry buffer (reduces bandwidth required).
Master (Rank 0): Scatter colonies to workers

For each colony:
    Worker (Rank N) including master:
        For each ant we have to process:
            Process the ant.
```

```
        Send master an updated vector of the colony states we worked on.
        Send master updated SnapGrids and which tiles were written.
    Master: Receive data from all workers.

Master: Serial code (update SnapGrids, colony work, etc).
```

The pseudocode brings up a few interesting design considerations, namely in how to serialise/deserialise data structures like the SnapGrid, and especially `std::vector<Colony`. The 2D SnapGrids (obstacles and food) are stored in a single contiguous array and are just of the `bool` datatype, so they are easy to transmit using MPI function calls like `MPI_Bcast`. The 3D SnapGrid (pheromone strength), however, is made up of `PheromoneStrength` structs, which cannot be directly transmitted. To send them over MPI, each of these `PheromoneStrength` structs was unpacked into its component parts: `toColony` and `toFood`, and transmitted as a flat array of doubles.

Unfortunately, it's also required to transmit the `std::vector<Colony>`. The `Colony` struct is a very complex datatype, with nested structs, so can't just be unpacked like the 3D SnapGrid. Instead, the C++ serialisation library Cereal was used to actually serialise and deserialise the entire struct to a buffer of bytes. These bytes are then transmitted using `MPI_Send`, and Cereal is used to deserialise the object on the receiving end of the program. This obviously has many performance considerations, but is the best method possible given the current design of the simulator.

In terms of how the work is distributed to the MPI workers, I chose to scatter the colonies rather than the individual ants themselves. Due to how `MPI_Scatter` is used in the code, one limitation is that the number of MPI workers must be divisible by the number of colonies. In case this assumption is not true, the program detects this and exits with an error.

Given the current design of MPI work distribution (the colonies are distributed), it should also be possible to merge OpenMP and MPI together, for even more performance. However, due to issues with OpenMP causing de-synchronisation, this was not attempted in this paper.

3.4 Verification

In the serial version, the goal of having the simulator be 100% deterministic between runs was achieved. Originally, this same goal was going to be used for the parallel implementation, and considerations like critical sections and making the PCG RNG thread-safe were taken into account. However, it was quickly discovered that this was not going to be possible - or at least, not without very significant performance overheads.

The main reason why determinism is impossible is to do with threading. Imagine two ants located nearby in the SnapGrid data structure. As it turns out, updating these ants will involve calling `pheromoneGrid.write(128, 128, ...);`, with differing pheromone values. In the serial version of the simulator, that ants are run one after the other, so this is not an issue and doesn't break determinism. However, in the OpenMP version of the simulator, these ant updates could be occurring in two separate threads at the exact same time instant. This was taken into account, and so all *writes* to the SnapGrid are surrounded in OpenMP critical sections, so only one thread can perform them at a time (the nature of SnapGrid means *reads* don't need to be protected). However, the big issue is that the direction in which the contention between these two threads is resolved is completely non-deterministic at the OS-level. In one run of the simulator, the first thread might get to write first: in another run, the second thread might get in. Because of this, and because we cannot control the Linux thread scheduler, unfortunately means that the multi-threaded simulator cannot be deterministic.

Since, unfortunately, the simulator results cannot be 100% deterministic as explained above, a different verification methodology for the parallel techniques was needed. To make sure the simulation mechanics weren't bugged by the OpenMP or MPI implementation, recordings of many different maps were made (including with different thread counts/worker counts), played back, and compared to the serial version. For OpenMP, although it didn't finish in the exact same state as the equivalent serial version, since the mechanics didn't change, it was considered "good enough", given the level of performance benefits observed. Unfortunately, the same could not be completely determined for MPI. Although the simulator still ran for a reasonable looking number of ticks, the rendering of the simulator appears to have been bugged. Food was still being eaten, but the ants eating it were not being rendered in the recording. Due to time constraints, this was assumed to be a graphical glitch (since the food was still being eaten), and was not addressed for this simulator - although it should be lookined into in the future.

3.5 Performance results

As with the serial benchmarks, performance of the simulator was measured using simulation time in milliseconds and simulation ticks per second. For this set of benchmarks, PNG TAR recording was enabled (in order to

verify and display the output). Particular attention was paid to how the simulator scales on a very large map with a huge number of ants, to see how effective each multi-threaded optimisation was.

To ensure the results are comparable, the batch job was restricted to run on nodes with the "R640" constraint, which at the time of writing are: smp-7-[0-4]. These nodes all have Intel Xeon Gold 6132 CPUs.

3.5.1 OpenMP benchmarks

The graph below (Figure 8) shows how OpenMP improves scalability with many ants compared to the serial implementation. It was generated using megamap which is 960x540 pixels and contains 743 colonies (see Appendix A). This is a significant increase over the amount of colonies used to benchmark the serial implementation previously, so should really be able to push the simulator code to its limits.

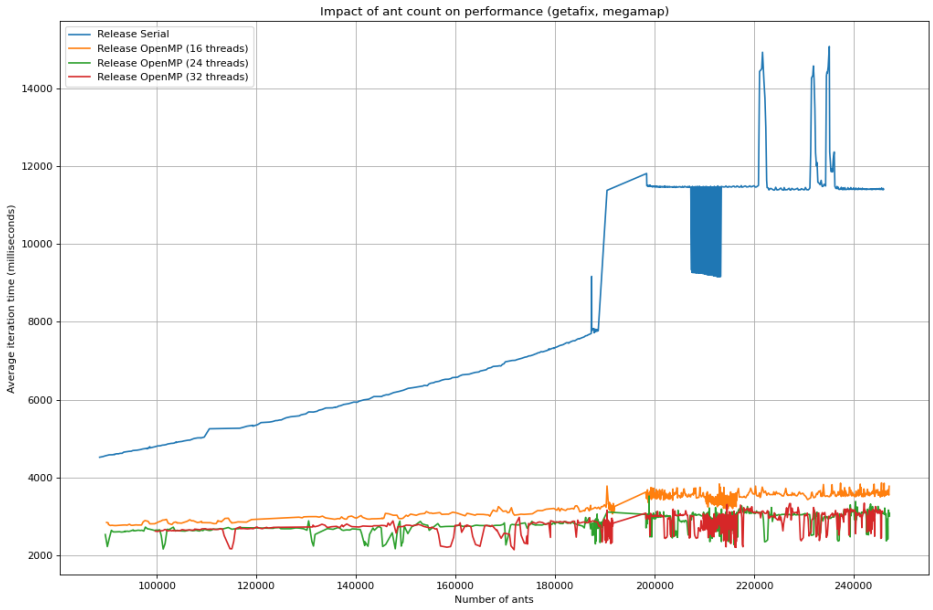

Figure 8: Graph demonstrating how the number of ants in the simulator affects the simulator update time in milliseconds, when using various numbers of OpenMP threads, compared against the serial implementation.

OpenMP dramatically improves the scalability of the simulator. The serial graph shows that the simulator slows down linearly as a function of the number of ants active at that time. The OpenMP graphs, however, are comparatively flat (with some noise). This means that adding more ants to the OpenMP version of the simulator adds almost no performance penalty at all. In addition, the graph proves that adding more OpenMP threads improves the simulator speed (compare 16 threads with 32 threads). Even 16 threads is significantly faster than the serial implementation. Overall, this is a great result and shows that the simulator performance has been improved.

In addition to the above graph, based on the same data and the same runtime conditions, the optimisation technique used was compared against the number of seconds it took for the entire simulation to run up to 2000 iterations. See Figure 9.

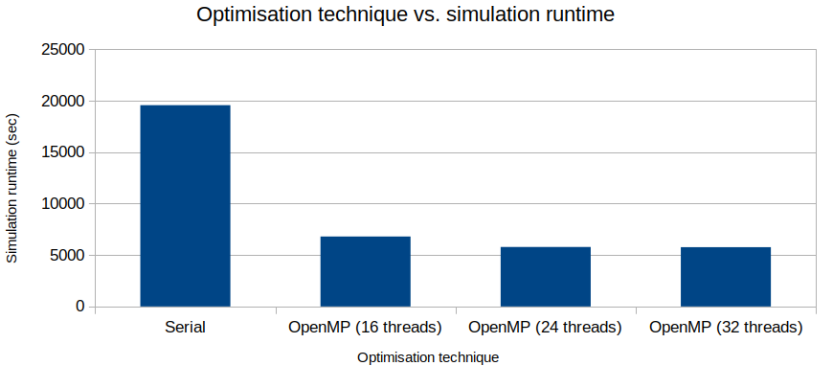

Figure 9: Graph comparing optimisation technique used vs. the overall time the simulator took to complete 2000 iterations.

This graph again shows how much OpenMP improves the performance of the simulator. Interestingly, however, there's not all that much difference between 16 threads and 32 threads in overall simulation runtime. It appears that just the "having threads" part is the most important, rather than how many of them there are.

3.5.2 MPI benchmarks

Unfortunately, while preparing benchmarks for MPI, multiple issues were encountered that made it very difficult. Firstly, significant memory limitations were discovered. Unfortunately, the simulator is so memory hungry that with MPI's multiple worker processes (not shared memory), it was simply impossible to run the usual megamap on getafix. Instead, a version with a significantly reduced number of colonies (20 colonies) was made called megamap_mpi. Because MPI is so slow, there was not enough time to run a full 2000 iteration simulation as was done previously, only enough time for 500 iterations. This in turn caused the map to not evolve enough to generate a number of ants vs. average iteration time graph, as was made for OpenMP and serial.

Instead, the following graph (Figure 10) was generated, which just contains the runtime of the megamap_mpi map compared against serial, OpenMP and MPI. It's still useful for comparison, but it's still unfortunate that so many limitations were encountered trying to benchmark MPI, and this is definitely something that should be improved in future. In the graph, labels like "20 workers/5 per node" mean that 20 MPI workers were used in total, and Slurm was instructed to allocate 5 workers for each node used (implying that 4 nodes total were used).

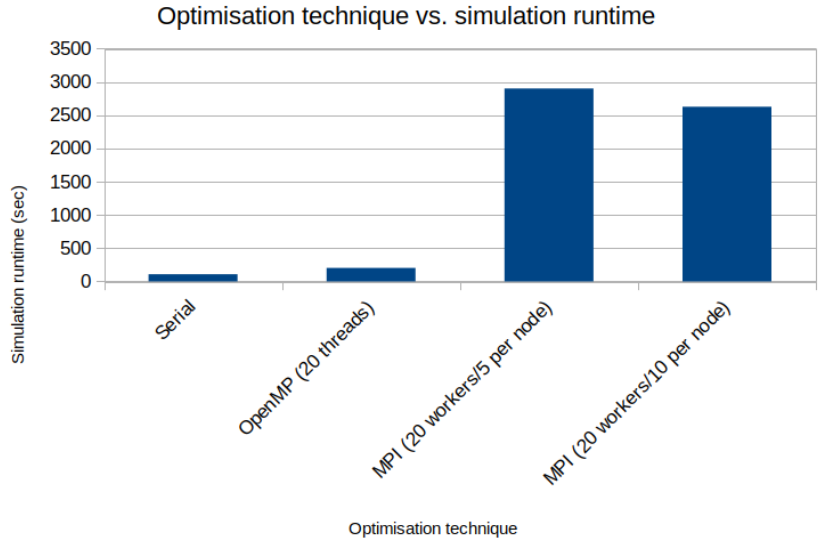

Figure 10: Graph comparing optimisation technique used vs. the overall time the simulator took to complete 500 iterations.

Certainly, the results here are not what we would be expecting. MPI performs incredibly poorly when compared to serial and OpenMP. In fact, according to the data, MPI with 5 workers per node is over 27x

slower than serial. Also interesting to note is that for this graph, OpenMP is actually slower than serial as well (although it's not very visible because of how slow MPI is). So, why is MPI so slow? I still believe that MPI has the potential to be fast, but because of the way I've designed the simulator, the amount of data that each worker has to copy back and forth between the master - not to mention the overhead of using serialisation frameworks like Cereal - creates significant slowdown. On large maps, each worker may be exchanging megabytes worth of data with the master, which even over a high-speed interconnect, is unacceptable. The copious use of barriers to ensure determinism also will not help performance. All of these factors combine together to create huge amounts of overhead that make it extremely slow.

3.6 Profiling

To profile both the OpenMP and MPI implementations, Valgrind's Callgrind and Cachegrind tools were used once again, along with KCachegrind to visualise the results.

3.6.1 OpenMP profiling

Using KCachegrind, a call graph for the OpenMP version of the program was rendered (see Figure 11). The same conditions as the serial profiling were used: the map used was map4 (128x128 with 4 colonies), and the machine used was my 16-core, 32-thread AMD Ryzen 9 5950X with 32GB of RAM. The number of OpenMP threads was set to 32.

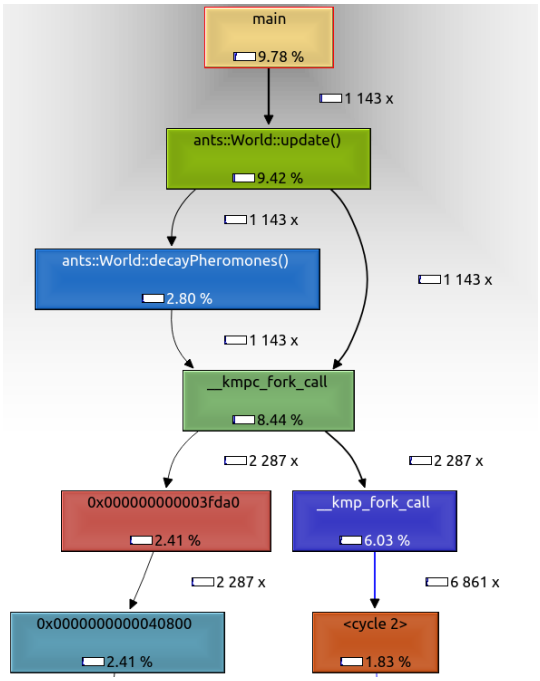

Figure 11: Call graph for 32-thread OpenMP simulator, KCachegrind.

Unfortunately, the results of this call graph are actually difficult to place precisely. The timings of each function are certainly very different from the serial call graph. For example, in the serial implementation, `World::decayPheromones` took 33.82% of the entire program's runtime, whereas now it only takes 2.80%. However, the tradeoff is it appears that there is a lot of overhead introduced by OpenMP (the `__kmpc_fork_call` function). Because OpenMP debug symbols are missing, the result of the call graph are just binary addresses that are unable to be analysed properly. From these results, it's clear that OpenMP appears to have made parallelised functions like `decayPheromones` take less time, at the cost of introducing some apparent thread spawning overhead.

3.6.2 MPI profiling

To profile MPI, the same conditions as all the profiling tasks so far were used. The only difference was that `mpiexec -n 4` (i.e. MPI with 4 workers) was used to generate the Cachegrind and Callgrind reports. This makes the full command to do profiling the following:

```
mpiexec -n 4 valgrind --tool=callgrind --cache-sim=yes --branch-sim=yes
./cmake-build-profile-llvm/ant_colony
```

The following call graph (Figure 12) shows the perspective of a 4-worker MPI master.

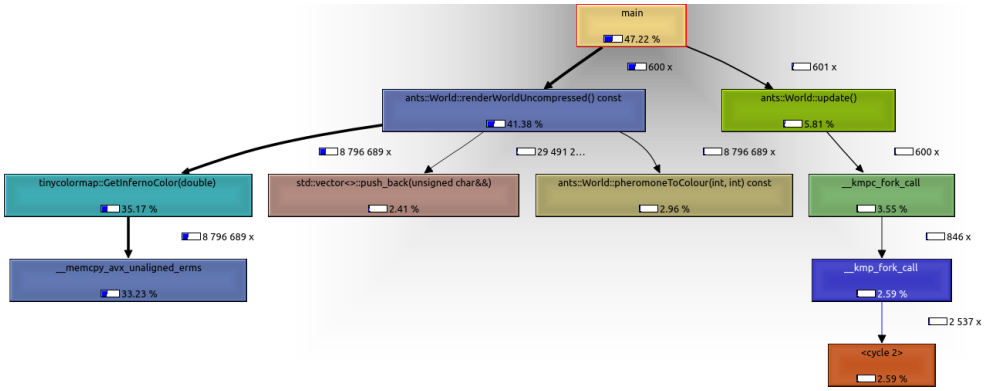

Figure 12: Call graph for 4-worker MPI simulator, master process, KCachegrind.

As can be seen above, the call graph looks quite different from all the other ones. In this case, the recording actually takes up the most time compared to `World::update` (recording was left in this run to record performance statistics). However, this does make sense: since there are 4 workers and 4 colonies, only 1 colony is being processed in the master.

3.7 Conclusion

The serial version of the simulator has been adapted and parallelised to run on high performance computing clusters using two techniques: OpenMP and MPI. Overall, both techniques significantly improve the performance and scalability of the simulator compared against the serial implementation. OpenMP can take advantage of multi-threading with the multiple cores of one HPC node, whereas MPI can distribute the work across the entire cluster.

4 Overall conclusion

In this paper, a high-performance, biologically inspired ant simulator has been presented. A serial version was written in C++17, and various features such as data serialisation into the "PNG TAR" file format were included. The serial version was deeply analysed and profiled using Valgrind's Callgrind and Cachegrind tools, to determine code hot spots for improvement. The simulator was verified using the recording outputs and benchmarked against a set of criteria. Additionally, the performance using different compiler optimisations and other statistics were presented.

The code was then was parallelised to run on high performance computing clusters using two different techniques: OpenMP and MPI. The grid data structure was rewritten to work while multi-threaded, into one of the key features of this report: the snapshot grid or "SnapGrid" data structure. The parallelised version of the simulator was verified against the original version, and the reason for any inconsistencies was explained (the Linux thread scheduler prevents perfect determinism). Finally, the results of each of the parallelisation techniques were compared against the serial version. OpenMP shows significantly better performance than serial for larger grid sizes, whereas MPI is unfortunately very slow, but if given more time, could probably be improved to be faster than OpenMP on sufficiently large HPC clusters.

The most important lesson learned completing this project are:

- Plan for parallelism, and specifically what technique will be used. While complex nested structs may work for OpenMP, they significantly complicate things for CUDA and MPI.
- Profile code early and often. As shown with `computePheromoneVector`, functions can sometimes surprise you with how slow (or fast!) they are.
- High-performance computing is all about trade-offs. For example, while OpenMP is faster for sufficiently large grid sizes, the overhead it introduces can actually make it slower on smaller grids.

5 Appendices

5.1 Appendix A: Simulator maps & additional photos

The following are the maps used in testing the simulator:

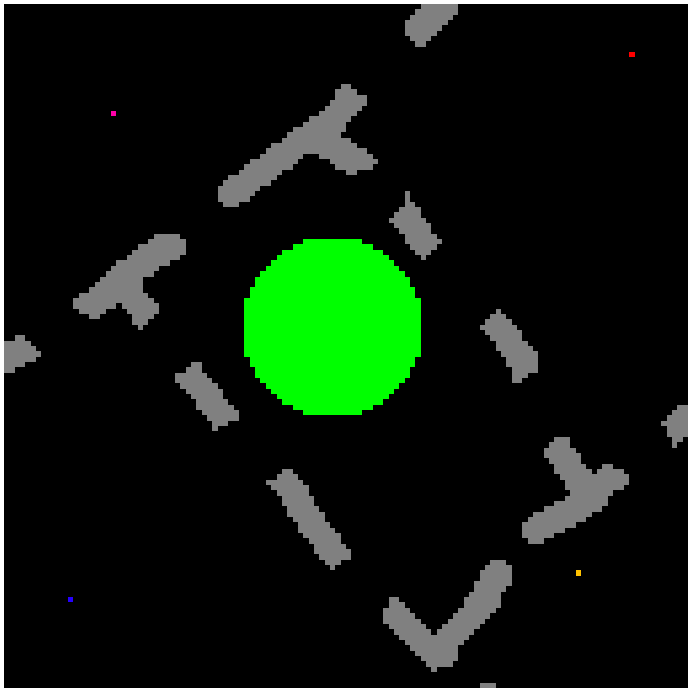

Figure 13: map4.png, 128x128, 4 colonies

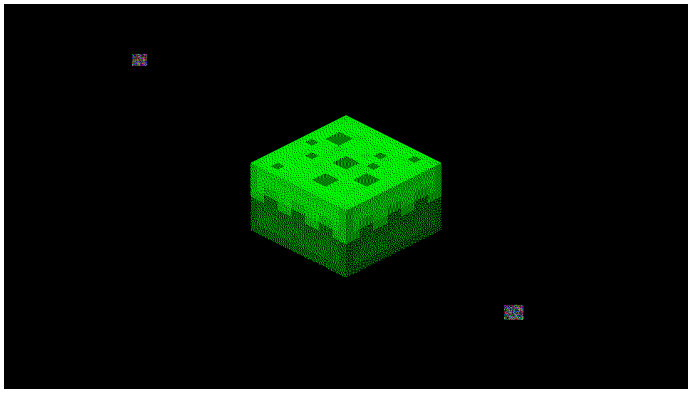

Figure 14: megamap.png, 960x540, 743 colonies

5.2 Appendix B: Open source libraries

This project makes use of the following open source libraries. All of the below libraries are released under permissive licences that allow their inclusion in the project. Special thanks go out to the authors of the following:

- stb_image (MIT licence): image loading library, used for map loading
- stb_image_write (MIT licence): image writing library, used for PNG TAR recording
- mINI (MIT licence): INI parsing library, used to load sim config
- microtar (MIT licence): TAR IO library, used for PNG TAR recording
- pcg-cpp (Apache 2.0 licence): C++ implementation of the high-quality PCG [9] RNG algorithm
- tinycolormap (MIT licence): C++ implementation of Matplotlib colour maps, used in PNG TAR recording
- clip (MIT licence): C++ clipboard ibrary, used only for development purposes
- cereal (BSD 3-clause licence): C++ binary serialisation/deserialisation, used for MPI communication

6 References

[1] Eric Bonabeau. "Agent-based modeling: Methods and techniques for simulating human systems". In: *Proceedings of the National Academy of Sciences* 99.suppl_3 (2002), pp. 7280–7287. DOI: 10.1073/pnas.082080899. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.082080899. URL: https://www.pnas.org/doi/abs/10.1073/pnas.082080899.

[2] Ashraf Darwish. "Bio-inspired computing: Algorithms review, deep analysis, and the scope of applications". In: *Future Computing and Informatics Journal* 3.2 (2018), pp. 231–246. ISSN: 2314-7288. DOI: https://doi.org/10.1016/j.fcij.2018.06.001. URL: https://www.sciencedirect.com/science/article/pii/S2314728818300631.

[3] Leonora Bianchi et al. "A survey on metaheuristics for stochastic combinatorial optimization". In: *Natural Computing* 8 (2008), pp. 239–287.

[4] Marco Dorigo and Thomas Stützle. "Ant Colony Optimization: Overview and Recent Advances". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Boston, MA: Springer US, 2010, pp. 227–263. ISBN: 978-1-4419-1665-5. DOI: 10.1007/978-1-4419-1665-5_8. URL: https://doi.org/10.1007/978-1-4419-1665-5_8.

[5] Pan Junjie and Wang Dingwei. "An Ant Colony Optimization Algorithm for Multiple Travelling Salesman Problem". In: *First International Conference on Innovative Computing, Information and Control - Volume I (ICICIC'06)* 1 (2006), pp. 210–213.

[6] Michalis Mavrovouniotis and Shengxiang Yang. "Ant colony optimization with immigrants schemes for the dynamic travelling salesman problem with traffic factors". In: *Appl. Soft Comput.* 13 (2013), pp. 4023–4037.

[7] Robin M. Weiss. "Chapter 22 - GPU-Accelerated Ant Colony Optimization". In: *GPU Computing Gems Emerald Edition*. Ed. by Wen-mei W. Hwu. Applications of GPU Computing Series. Boston: Morgan Kaufmann, 2011, pp. 325–340. ISBN: 978-0-12-384988-5. DOI: https://doi.org/10.1016/B978-0-12-384988-5.00022-X. URL: https://www.sciencedirect.com/science/article/pii/B978012384988500022X.

[8] Sebastiano Vigna. "It is high time we let go of the Mersenne Twister". In: *CoRR* abs/1910.06437 (2019). arXiv: 1910.06437. URL: http://arxiv.org/abs/1910.06437.

[9] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. rep. HMC-CS-2014-0905. Claremont, CA: Harvey Mudd College, Sept. 2014.

[10] Kostya Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX Annual Technical Conference*. 2012.

[11] Simon Byrne. "Beware of fast-math". Nov. 2021. URL: https://simonbyrne.github.io/notes/fastmath/.

[12] Alwyn Husselmann and Kenneth A. Hawick. "Spatial Data Structures, Sorting and GPU Parallelism for Situated-agent Simulation and Visualisation". In: 2012.