University of Edinburgh

Programming Skills

Performance Testing Coursework

Performance Report

Author:
Jack Frankland

November 29, 2017

Abstract

This report provides the results and analysis of various performance tests which were carried out on the 2D Predator-Prey model that was written for the first coursework. A range of different tests are used to evaluate the program and in each case the tests results are analyzed and interpreted. A conclusion is then reached about the overall quality of the program in terms of performance.

List of Figures

List of Tables

Contents

1 Introduction

In order to provide a comprehensive performance analysis of the predator-prey simulation program we wrote, this report focuses on three different areas of testing. We begin by examining the effects of compiling the program under various compilers and their respective optimization flags on several different computers. This allows us to determine under which compiler and at which optimization level the program runs quickest. We then proceed to profile the program using a profiler tool. By identifying in which functions calls are the most computationally expensive we are able to identify the main overheads in our code, and in recognizing these overheads we can suggest possible optimizations that could result in faster code. Once we have finished investigating the performance of the code itself, we turn our attention to the performance of the code in terms of input and output. During its run the program outputs plain .ppm files at a frequency specified by the user; we investigate the performance effect of varying this frequency. Finally we test the code in terms of input by running it first with a range of landscapes with different land:water ratios, then with a range of landscapes of different sizes. This completes are performance testing, and based on our results we then make a general conclusion about the overall performance of the code.

Most of the performance tests in this document involve recording the runtime of the code for various independent variables. In order to get an accurate figure on the run time of the code in each test we initially planned to run the tests on the a single core on a Cirrus back end node. Because the traffic to Cirrus is fairly consistent, due to the fact is in constant use, this would have allowed us to make performance measurements we know are reconstructible, and hence could have compared run times with a good degree of confidence in their relative values. However, after some initial testing it was clear the code ran roughly 4-5 times faster on the cp-lab than it did on Cirrus. Further to this, we found that submitting several jobs to the Cirrus queue had the effect of slowing each job down by an amount proportional to the number of jobs submitted. The reason for this is not entirely clear; one possibility is that the different jobs were competing for memory on the front end node during output. This issue in particular made testing on Cirrus infeasible, since some run times were well over ten minutes, and the jobs needed to be submitted one at a time, the total runtime would have been too large to get any results. Instead we decided to use the cp-lab, since the run times were faster we were able to submit jobs individually and since the cp-lab is in use most of the time we can assume our results accurately represent a standard job runtime.

2 Performance Tests and Analysis

2.1 Effects of Compiler Optimization Flags

In this section we will investigate the performance effects of using different compilers with different optimization flags. To implement these tests we use a single 1000×1000 landscape given in figure 1 which consists of two large land masses separated by a river of water. This landscape was chosen since it is large enough and generic enough to provide results that could be scaled for any realistic landscape. The input parameters for all tests in this section were fixed as the values given in appendix C.

We ran 6 tests for each compiler flag and calculated the mean and stan-

Figure 1: Generic Test Landscape.

dard error using equations 1 and 2. A complete set of measurements is given in A. For an explanation of what the various compiler optimization levels mean run any of `man clang++`, `man g++` or `man icpc` on a machine with the respective compiler installed and read the section on optimization flags.

Compiler	**Run Time(s)**					
	O0	**O1**	**O2**	**O3**	**Ofast**	**Os**
gcc	249.002 ± 0.765	181.647 ± 0.571	172.332 ± 0.912	174.978 ± 0.777	177.254 ± 1.127	190.362 ± 0.496

Table 1: Run times for the 1000×1000 landscape in figure 1 running on the cp-lab.

From these results it is clear that under the GNU g++ compiler the code runs fastest when compiled under the $\mathcal{O}2$ flag. This is slightly unexpected since one would expect the code to run fastest when compiled under the $\mathcal{O}3$ or $\mathcal{O}$fast flags, however it is fairly common for different programs to run faster under different compiler flags due to the different optimisations the various compiler flags employ, and the content of the code itself.

We were interested in how well the program would perform under a completely different compiler, which is one of the reason we initially chose to use Cirrus. However, due to the fact the the cp-lab only has the GNU g++

compiler installed this meant using a different machine to compile the code. We chose to carry out these extra experiments on a Cirrus back end node and a Macbook to see how they would compare. Due to the long runtime for each test we were only able to carry out one test in each case and have not performed any statistical averages.

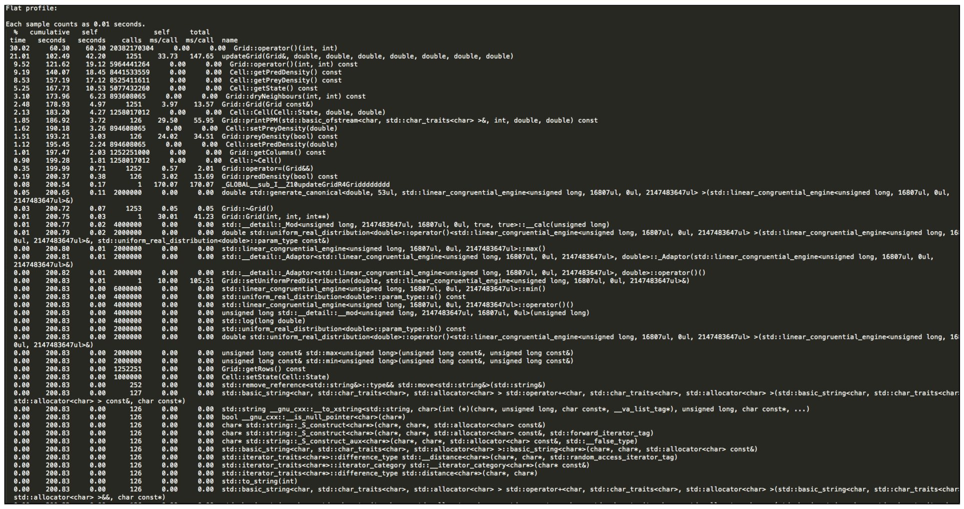
Compiler	**Run Time(s)**					
	O0	**O1**	**O2**	**O3**	**Ofast**	**Os**
clang	529.025	457.535	395.897	415.884	383.270	399.656
intel	552.608	378.542	373.768	361.429	687.909	395.915

Table 2: Run times for the 1000 $\times$ 1000 landscape in figure 1 running on mid 2014 Macbook Pro with 2.8GHz Intel i5 processor with 8 GB 1600 MHz DDR3 memory.

Compiler	**Run Time(s)**					
	O0	**O1**	**O2**	**O3**	**Ofast**	**Os**
gcc	800.220	723.293	724.042	698.490	698.005	734.928
intel	871.976	751.004	728.728	718.015	723.325	731.965

Table 3: Run times for the 1000 $\times$ 1000 landscape in figure 1 running on Cirrus back end node with 1 cpu.

From tables 2 and 3 we can see the Intel compiler produces an executable that runs fastest (relative to the machine it is compiled on) at the $\mathcal{O}3$ optimisation level. Since in the case of Cirrus the Intel compiler produces a program that runs faster than the GNU compiler, and on the Macbook the same Intel compiler produces a program that runs faster than the Clang compiler, this suggests that our program performs best when compiled under the Intel compiler. If the cp-lab had the Intel compiler suite installed it would be a good test to see how the run times of our program compared to the g++ results in table 1 when compiled under the Intel icpc compiler. However, since the above results are not statistical averages it is difficult to make any definitive statement about any relative performance increase of code compiler under the Intel compiler; more results would need to be measured before any



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 30.02     60.30    60.30 20382170304     0.00     0.00  Grid::operator()(int, int)
 21.01    102.49    42.20     1251    33.73   147.65  updateGrid(Grid&, double, double, double, double, double, double, double)
  9.52    121.62    19.12 5964441264     0.00     0.00  Grid::operator()(int, int) const
  9.19    140.07    18.45 8441533559     0.00     0.00  Cell::getPredDensity() const
  8.53    157.19    17.12 8525411611     0.00     0.00  Cell::getPreyDensity() const
  5.25    167.73    10.53 5077432260     0.00     0.00  Cell::getState() const
  3.10    173.96     6.23 893608065     0.00     0.00  Grid::dryNeighbours(int, int) const
  2.48    178.93     4.97     1251     3.97    13.57  Grid::Grid(Grid const&)
  2.13    183.20     4.27 1258017012     0.00     0.00  Cell::Cell(Cell::State, double, double)
  1.85    186.92     3.72      126    29.50    55.95  Grid::printPPM(std::basic_ofstream<char, std::char_traits<char> >&, int, double, double) const
  1.62    190.18     3.26 894608065     0.00     0.00  Cell::setPreyDensity(double)
  1.51    193.21     3.03      126    24.02    34.51  Grid::preyDensity(bool) const
  1.12    195.45     2.24 894608065     0.00     0.00  Cell::setPredDensity(double)
  1.01    197.47     2.03 1252251000     0.00     0.00  Grid::getColumns() const
  0.90    199.28     1.81 1258017012     0.00     0.00  Cell::~Cell()
  0.35    199.99     0.71     1252     0.57     2.01  Grid::operator=(Grid&&)
  0.19    200.37     0.38      126     3.02    13.69  Grid::predDensity(bool) const
  0.08    200.54     0.17        1   170.07   170.07  _GLOBAL__sub_I__Z10updateGridR4Griddddddd
  0.05    200.65     0.11  2000000     0.00     0.00  double std::generate_canonical<double, 53ul, std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul> >(std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 
  2147483647ul>&)
  0.03    200.72     0.07     1253     0.05     0.05  Grid::~Grid()
  0.01    200.75     0.03        1    30.01    41.23  Grid::Grid(int, int, int**)
  0.01    200.77     0.02  4000000     0.00     0.00  std::__detail::_Mod<unsigned long, 2147483647ul, 16807ul, 0ul, true, true>::__calc(unsigned long)
  0.01    200.79     0.02  2000000     0.00     0.00  double std::uniform_real_distribution<double>::operator()<std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul> >(std::linear_congruential_engine<unsigned long, 16
  0ul, 2147483647ul>&, std::uniform_real_distribution<double>::param_type const&)
  0.00    200.80     0.01  2000000     0.00     0.00  std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>::max()
  0.00    200.81     0.01  2000000     0.00     0.00  std::__detail::_Adaptor<std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>, double>::_Adaptor(std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 
  2147483647ul>&)
  0.00    200.82     0.01  2000000     0.00     0.00  std::__detail::_Adaptor<std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>, double>::operator()()
  0.00    200.83     0.01        1    10.00   105.51  Grid::setUniformPredDistribution(double, std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>&)
  0.00    200.83     0.00  6000000     0.00     0.00  std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>::min()
  0.00    200.83     0.00  4000000     0.00     0.00  std::uniform_real_distribution<double>::param_type::a() const
  0.00    200.83     0.00  4000000     0.00     0.00  std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>::operator()()
  0.00    200.83     0.00  4000000     0.00     0.00  unsigned long std::__detail::__mod<unsigned long, 2147483647ul, 16807ul, 0ul>(unsigned long)
  0.00    200.83     0.00  4000000     0.00     0.00  std::log(long double)
  0.00    200.83     0.00  2000000     0.00     0.00  std::uniform_real_distribution<double>::param_type::b() const
  0.00    200.83     0.00  2000000     0.00     0.00  double std::uniform_real_distribution<double>::operator()<std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul> >(std::linear_congruential_engine<unsigned long, 16
  0ul, 2147483647ul>&)
  0.00    200.83     0.00  2000000     0.00     0.00  unsigned long const& std::max<unsigned long>(unsigned long const&, unsigned long const&)
  0.00    200.83     0.00  2000000     0.00     0.00  unsigned long const& std::min<unsigned long>(unsigned long const&, unsigned long const&)
  0.00    200.83     0.00  1252251     0.00     0.00  Grid::getRows() const
  0.00    200.83     0.00  1000000     0.00     0.00  Cell::setState(Cell::State)
  0.00    200.83     0.00      252     0.00     0.00  std::remove_reference<std::string&>::type&& std::move<std::string&>(std::string&)
  0.00    200.83     0.00      127     0.00     0.00  std::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_traits<char>, std::allocator<char> >(std::basic_string<char, std::char_traits<char
  std::allocator<char> > const&, char const*)
  0.00    200.83     0.00      126     0.00     0.00  std::string __gnu_cxx::__to_xstring<std::string, char>(int (*)(char*, unsigned long, char const*, __va_list_tag*), unsigned long, char const*, ...)
  0.00    200.83     0.00      126     0.00     0.00  bool __gnu_cxx::__is_null_pointer<char>(char*)
  0.00    200.83     0.00      126     0.00     0.00  char* std::string::_S_construct<char*>(char*, char*, std::allocator<char> const&)
  0.00    200.83     0.00      126     0.00     0.00  char* std::string::_S_construct<char*>(char*, char*, std::allocator<char> const&, std::forward_iterator_tag)
  0.00    200.83     0.00      126     0.00     0.00  char* std::string::_S_construct_aux<char*>(char*, char*, std::allocator<char> const&, std::__false_type)
  0.00    200.83     0.00      126     0.00     0.00  std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string<char*>(char*, char*, std::allocator<char> const&)
  0.00    200.83     0.00      126     0.00     0.00  std::iterator_traits<char*>::difference_type std::__distance<char*>(char*, char*, std::random_access_iterator_tag)
  0.00    200.83     0.00      126     0.00     0.00  std::iterator_traits<char*>::iterator_category std::__iterator_category<char*>(char* const&)
  0.00    200.83     0.00      126     0.00     0.00  std::iterator_traits<char*>::difference_type std::distance<char*>(char*, char*)
  0.00    200.83     0.00      126     0.00     0.00  std::to_string(int)
  0.00    200.83     0.00      126     0.00     0.00  std::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_traits<char>, std::allocator<char> >(std::basic_string<char, std::char_traits<char
  std::allocator<char> >&&, char const*)
```

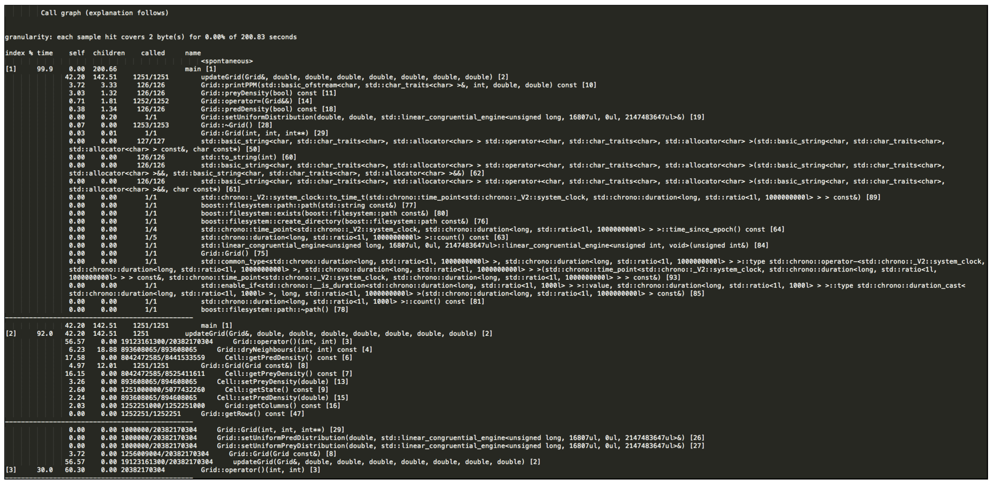
Figure 2: Flat profile page 1. for GNU gprof profiler.

conclusions could be reached. The relative decrease in run time both in the case of the Macbook (compared to the Clang compiler) and Cirrus (compared to the GNU compiler), suggest there may be some performance bonus to compiling under the Intel compiler.

2.2 Main Sources of Overhead

In order to identify the main sources of overhead in our program we used the GNU gprof tool. The code was compiled under the g++ compiler with optimisation level $\mathcal{O}$ (i.e. no optimizations) and without the debugging flags. This is because compiling with optimizations turned on can lead to inlining of functions, which in turn causes them to disappear from the profiling tables. However, since different compilers and different flags may inline different functions in different places, the profiling data would be specific to the compiler case, and not a general profile of the code. By compiling at $\mathcal{O}0$ level we can identify overheads that will be common to all compilers when there is no optimization. The input parameters and landscape are the same as in section 2.1. Since they are extensive and rather verbose we have included the first pages only since they contain the functions with the largest overheads.

The *flat profile* shows the total amount of time the program spent executing each function. From figure 2 we can see the program spent the most time executing the `operator()`. This is to be expected since the `operator()` func-



```
                Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.00% of 200.83 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.9    0.00  200.66                 main [1]
               42.20  142.51    1251/1251        updateGrid(Grid&, double, double, double, double, double, double, double) [2]
                3.72    3.33     126/126         Grid::printPPM(std::basic_ofstream<char, std::char_traits<char> >&, int, double, double) const [10]
                3.03    1.32     126/126         Grid::preyDensity(bool) const [11]
                0.71    1.81    1252/1252        Grid::operator=(Grid&&) [14]
                0.38    1.34     126/126         Grid::predDensity(bool) const [18]
                0.00    0.20       1/1           Grid::setUniformDistribution(double, double, std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>&) [19]
                0.07    0.00    1253/1253        Grid::~Grid() [28]
                0.03    0.01       1/1           Grid::Grid(int, int, int**) [29]
                0.00    0.00     127/127         std::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_traits<char>, std::allocator<char> >(std::basic_string<char, std::char_traits<char>,
                std::allocator<char> > const&, char const*) [50]
                0.00    0.00     126/126         std::to_string(int) [60]
                0.00    0.00     126/126         std::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_traits<char>, std::allocator<char> >(std::basic_string<char, std::char_traits<char>,
                std::allocator<char> >&&, std::basic_string<char, std::char_traits<char>, std::allocator<char> >&&) [62]
                0.00    0.00     126/126         std::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_traits<char>, std::allocator<char> >(std::basic_string<char, std::char_traits<char>,
                std::allocator<char> >&&, char const*) [61]
                0.00    0.00       1/1           std::chrono::_V2::system_clock::to_time_t(std::chrono::time_point<std::chrono::_V2::system_clock, std::chrono::duration<long, std::ratio<1l, 1000000000l> > > const&) [89]
                0.00    0.00       1/1           boost::filesystem::path::path(std::string const&) [77]
                0.00    0.00       1/1           boost::filesystem::exists(boost::filesystem::path const&) [80]
                0.00    0.00       1/1           boost::filesystem::create_directory(boost::filesystem::path const&) [76]
                0.00    0.00       1/4           std::chrono::time_point<std::chrono::_V2::system_clock, std::chrono::duration<long, std::ratio<1l, 1000000000l> > >::time_since_epoch() const [64]
                0.00    0.00       1/5           std::chrono::duration<long, std::ratio<1l, 1000000000l> >::count() const [63]
                0.00    0.00       1/1           std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>::linear_congruential_engine<unsigned int, void>(unsigned int&) [84]
                0.00    0.00       1/1           Grid::Grid() [75]
                0.00    0.00       1/1           std::common_type<std::chrono::duration<long, std::ratio<1l, 1000000000l> >, std::chrono::duration<long, std::ratio<1l, 1000000000l> > >::type std::chrono::operator-<std::chrono::_V2::system_clock,
                std::chrono::duration<long, std::ratio<1l, 1000000000l> >, std::chrono::duration<long, std::ratio<1l, 1000000000l> > >(std::chrono::time_point<std::chrono::_V2::system_clock, std::chrono::duration<long, std::ratio<1l,
                1000000000l> > > const&, std::chrono::time_point<std::chrono::_V2::system_clock, std::chrono::duration<long, std::ratio<1l, 1000000000l> > > const&) [93]
                0.00    0.00       1/1           std::enable_if<std::chrono::__is_duration<std::chrono::duration<long, std::ratio<1l, 1000l> > >::value, std::chrono::duration<long, std::ratio<1l, 1000l> > >::type std::chrono::duration_cast<
                std::chrono::duration<long, std::ratio<1l, 1000l> >, long, std::ratio<1l, 1000000000l> >(std::chrono::duration<long, std::ratio<1l, 1000000000l> > const&) [85]
                0.00    0.00       1/1           std::chrono::duration<long, std::ratio<1l, 1000l> >::count() const [81]
                0.00    0.00       1/1           boost::filesystem::path::~path() [78]
-----------------------------------------------
               42.20  142.51    1251/1251        main [1]
[2]     92.0   42.20  142.51    1251         updateGrid(Grid&, double, double, double, double, double, double, double) [2]
               56.57    0.00 19123161300/20382170304     Grid::operator()(int, int) [3]
                6.23   18.88 893608065/893608065     Grid::dryNeighbours(int, int) const [4]
               17.58    0.00 8042472585/8441533559     Cell::getPredDensity() const [6]
                4.97   12.01    1251/1251        Grid::Grid(Grid const&) [8]
               16.15    0.00 8042472585/8525411611     Cell::getPreyDensity() const [7]
                3.26    0.00 893608065/894608065     Cell::setPreyDensity(double) [13]
                2.60    0.00 1251000000/5077432260     Cell::getState() const [9]
                2.24    0.00 893608065/894608065     Cell::setPredDensity(double) [15]
                2.03    0.00 1252251000/1252251000     Grid::getColumns() const [16]
                0.00    0.00 1252251/1252251     Grid::getRows() const [47]
-----------------------------------------------
                0.00    0.00 1000000/20382170304     Grid::Grid(int, int, int**) [29]
                0.00    0.00 1000000/20382170304     Grid::setUniformPredDistribution(double, std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>&) [26]
                0.00    0.00 1000000/20382170304     Grid::setUniformPreyDistribution(double, std::linear_congruential_engine<unsigned long, 16807ul, 0ul, 2147483647ul>&) [27]
                3.72    0.00 1256009004/20382170304     Grid::Grid(Grid const&) [8]
               56.57    0.00 19123161300/20382170304     updateGrid(Grid&, double, double, double, double, double, double, double) [2]
[3]     30.0   60.30    0.00 20382170304         Grid::operator()(int, int) [3]
-----------------------------------------------
```

Figure 3: Call graph page 1. for GNU gprof profiler.

tion is used as a utility for accessing the landscape cells in the form `(i,j)`. Due to the fact that internally the landscape is implemented as a 1-D array this function was introduced to make working with the landscape simple and intuitive, and any time access to the landscape is required in the program, the `operator()` will have been used. One way to reduce the number of function calls to `operator()` would be to access the elements of the landscape directly in the code using the `operator[]` on the array of cells. This would make the code for the various Grid methods significantly harder to write and read since all instances of `operator(i,j)` would become `operator[[i + (rows + 1 - j) * (columns+2)]]`. An alternative would be to implement the landscape as a 2D array, then the access would be implemented via `operator[i][j]`, however it is likely the overhead of dynamically allocating a 2D array of a user defined type would outweigh the cost of the function we currently have.

The *call graph* shows how much time was spent in each function and its children. Form this information we are able to identify functions that themselves do not use much time, but call other functions that use exceptional amounts of time. From figure 3 line [2] we can see that 92% of the time was spent in the function `updateGrid()` and its subroutines, which is to be expected since this is the function that evolves the landscape and apart from printing output, most the remaining functions called from the main method perform one off tasks such as setting up initial conditions. Of the function calls from within `updateGrid()` we can see that `operator()` clocks in the highest with 56.57s spent within it when it is called from `updateGrid()`. This supports our above analysis that the function `operator()` is a serious overhead. From [3] and [5] (not shown in figure 3) we can see that 39.5% of the total time is spent in `operator()` or its constant equivalent. We can also see that it is the methods of the Grid class and the `updateGrid()` functions that call the `operator()`. Based on this analysis it would be a good idea to test giving these functions direct access to the grid cells through the `operator[]` instead of using the overloaded `operator()`. It may be that moving the functionality from `operator()` to `operator[]` means that the functions that call it inherit the overhead instead, however it is worth checking as a possible optimization.

2.3 Effects of Input/Output

2.4 Output Frequency

To begin the investigation into the effects of input we vary the number of time steps between the output of the plain .ppm files. The other input parameters are kept fixed at the same values in appendix C, however the number of steps between the output of the plain .ppm files is varied from 10 to 100 at intervals of 10 steps. The input landscape in figure 1 was used for the same reasons as in section 2.1. Since the program was shown to run fastest when compiled at the $\mathcal{O}2$ level in section 2.1, we have used that option here.

As in section 2.1 we took six measurements for each output frequency and calculated the error using equations 1 and 2. The error bars have been including in figure 4, however they are very small. A complete set of measurements is given in A. It is clear from figure 4 that the runtime increases dramatically for higher frequencies, where as for lower frequencies the runtime increase is a lot slower. This relationship is perhaps made clearer in figure 5 where we have plotted the same run times against the total number of output files. The plot is clearly linear, which is what we would expect; the total runtime should be directly proportional to the number of files it outputs. The total number of output files can be calculated by dividing the total number of time steps by the frequency and then flooring the number.

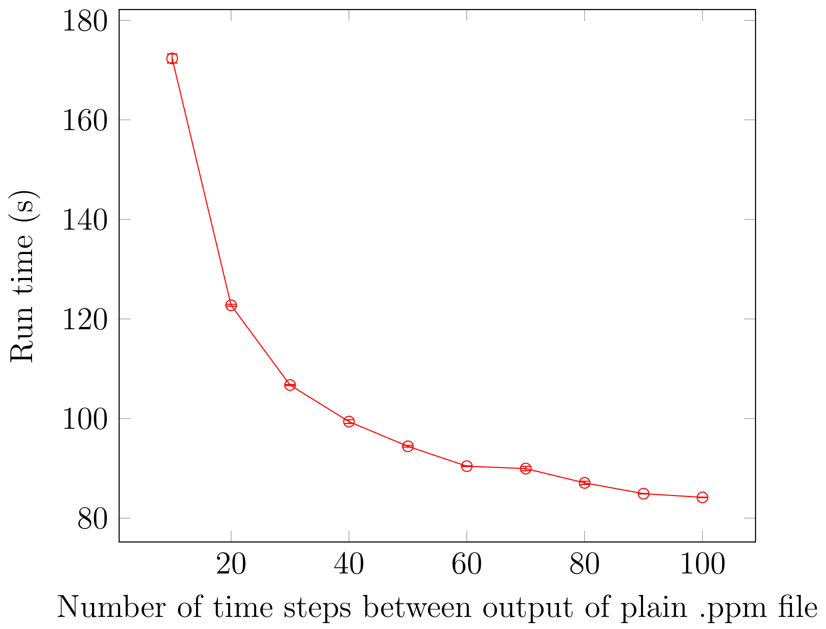


Figure 4: Output frequency of plain .ppm vs run time (s)

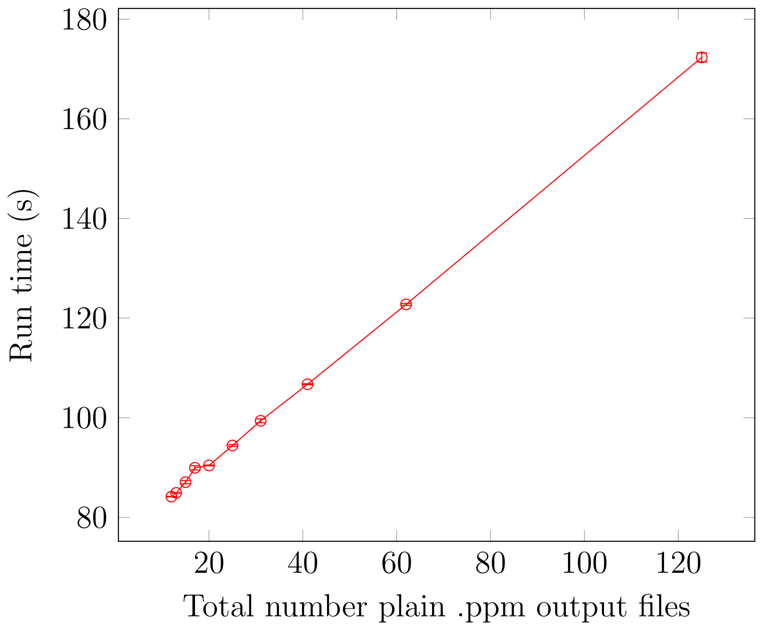
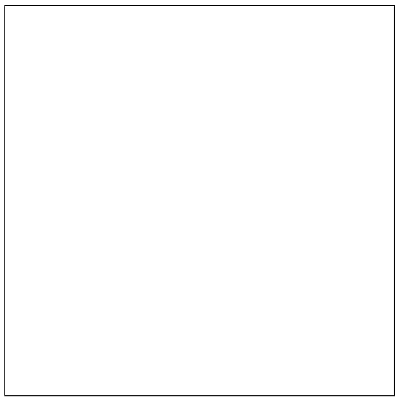


Figure 5: Number of output plain .ppm vs run time (s)

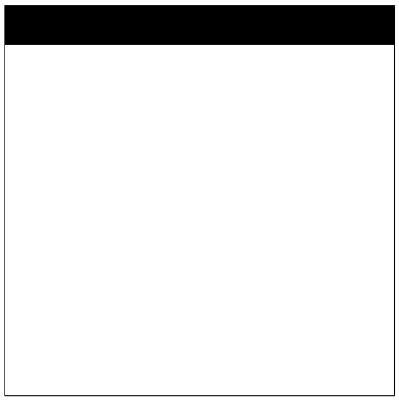
2.4.1 Landscape Water vs. Land

There are a myriad of possible ways to test the effects of varying the ratio of water to land in an input landscape. It is important to test the performance of the program using realistic input. For example, it would be possible to produce a landscape which was 50% land 50% water, where each pixel was alternatively water and land. Whilst this would be a valid input test landscape, it would not be a very good one, since it is so unrealistic it is unlikely anyone would ever try and model the evolution of a predator-prey system on a landscape of this sort. For these reasons we choose to test landscapes where land masses were always connected. In particular, to test the performance of the program under variation of the ratio of land to water in the input landscape we begun with the empty 1000×1000 landscape in 6a (i.e. all water) and introduced strips of land of dimension 1000×100 (10% of the total landscape) e.g. figures 6b and 6c. By placing these strips underneath one another we produced eleven different landscapes each one having 10% more land than the previous one, and eventually producing a landscape that was 100% land i.e. figure 6d. As in previous sections, we took six measurements at each land:water ratio and calculated errors using equations 1 and 2 and to shorten overall test times only output .ppm files every 100 time steps. A complete set of measurements is given in A. Again, errors have been plotted in figure 7, however they are very small. From section 2.4 we know that the runtime scales linearly with the number of output files produced. Thus in order to run quicker tests for the landscape size we ran with a low output frequency of 100 and assumed that for any other output frequency the result would scale linearly. All other input parameters are fixed as in appendix C.

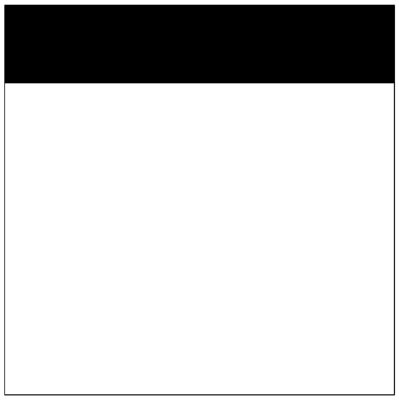
From figure 7 we can see that the run time scales linearly as the percentage of land in the landscape is increased. This is to be expected. Since the function `updateGrid()` first checks a cell in the landscape is water before updating it, and if it is water it ignores that cell; if we increase the number of land cells by say 10% we increase the number of cells `updateGrid()` has to update on each call by 10%, and since each cell is roughly the same, we would expect the time taken to complete the update to increase linearly as a function of the number of cells needing to be updated, i.e. the percentage of the landscape that is land.



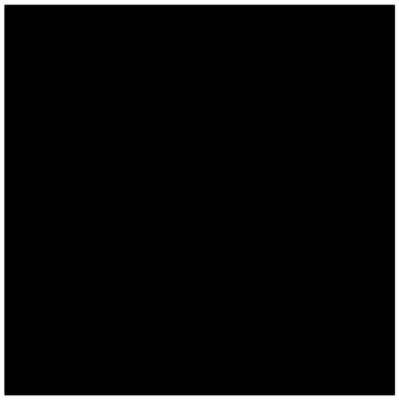
(a) Initial landscape, 0% land.



(b) 2nd landscape with 10% land.



(c) 3rd landscape with 20% land.



(d) Final landscape with 100% land.

Figure 6: Examples of test landscapes for the water vs. land tests.

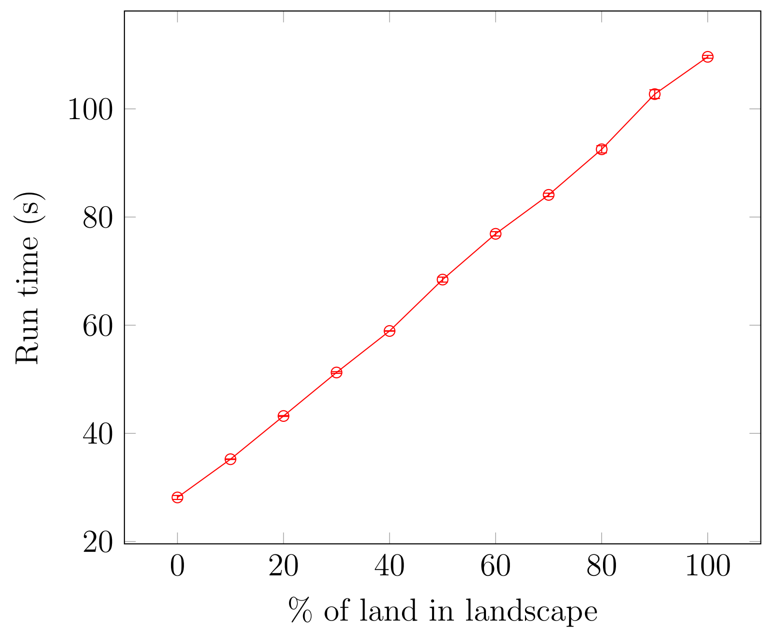


Figure 7: Water vs land runtime

2.4.2 Landscape Size

To investigate the effects of varying landscape size for our program we scaled the square landscape of figure 1 using Gimp by increasing the width and height be equal amounts and recorded the time take to run in each case. For the same reasons as above, we ran with a low output frequency of 100. As above we recorded six measurements at each landscape size and calculated standard error using equations 1 and 2. A complete set of measurements is given in A. The results plotted in 8 suggest a polynomial relationship between the size of the landscape and the run time of the form run time $= a(\text{landscape size})^k$. By taking the ln (natural log) of the data and plotting it in figure 9 we can approximate the parameters a and k above by measuring the y-intercept and the gradient respectively. We can read off figure 9 $a \approx 2.718$ and $k \approx 2$, so we get the relationship run time $= 2.718(\text{landscape size})^2$ (for a complete explanation of this calculation see **??**. The relationship run time $\propto (\text{landscape size})^2$ is to be expected, since the total area increases as the square of its sides for a square landscape, and we expect the run time to be proportional to the total area i.e. the number of squares in the landscape.

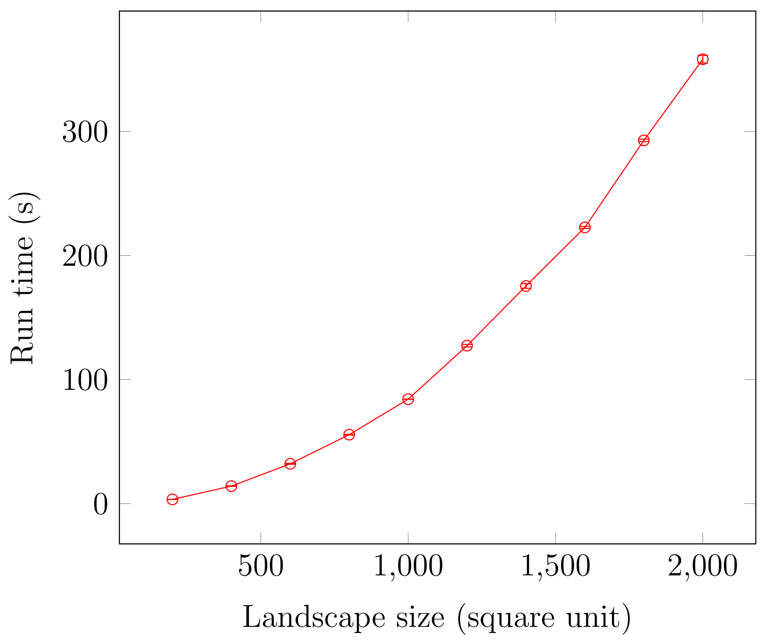


Figure 8: Size of square landscape vs run time (s)

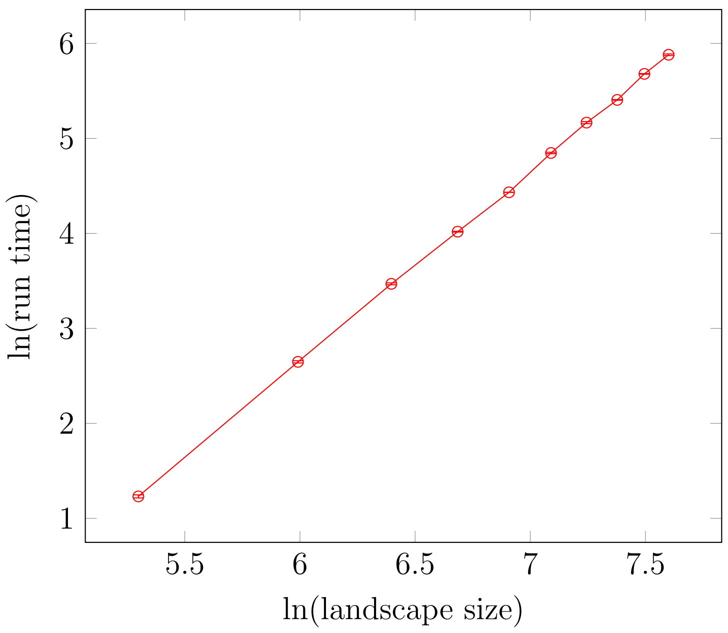


Figure 9: Log plot of size of square landscape vs run time (s)

3 Conclusion

A Complete Data Sets

These are the complete data sets for the performance experiments above, since tables 2 and 3 are complete, they are not repeated here:

Compiler	**Run Time(s)**					
	O0	O1	O2	O3	Ofast	Os
gcc	247.042	181.057	171.405	171.405	177.838	190.849
	247.466	181.326	171.641	175.660	176.470	190.547
	247.391	183.946	176.876	177.094	181.324	191.942
	250.712	182.675	171.137	175.355	179.191	190.994
	250.511	180.602	171.369	174.871	174.351	188.759
	250.888	180.273	171.561	175.481	174.350	189.081

Table 4: Complete data set for gcc optimisation tests on cp lab.

Number steps	**Run Time(s)**					
10	171.405	171.641	176.876	171.137	171.369	171.561
20	122.466	122.400	123.026	122.703	123.573	122.341
30	106.718	106.978	106.254	106.518	106.862	107.034
40	100.035	98.694	98.721	101.000	98.802	99.043
50	94.207	94.208	93.995	94.274	94.611	95.233
60	90.761	90.289	90.335	90.298	90.570	90.307
70	89.597	89.076	91.979	89.224	90.130	89662
80	88.188	87.924	86.422	87.013	86.457	86.422
90	84.877	84.799	84.838	84.877	84.945	85.100
100	84.151	84.138	84.095	84.392	84.051	84.146

Table 5: Complete data set for varying number of steps between output of plain .ppm.

Landscape Size	**Run Time(s)**					
200 × 200	3.372	3.409	3.711	3.366	3.3379	3.373
400 × 400	14.431	13.768	14.803	14.029	13.848	13.881
600 × 600	33.229	32.582	31.024	31.608	32.466	31.779
800 × 800	55.585	56.726	55.295	55.303	54.997	55.709
1000 × 1000	84.151	84.138	84.095	84.392	84.051	84.146
1200 × 1200	126.580	131.212	125.095	127.586	124.953	128.426
1400 × 1400	182.386	177.712	174.942	176.624	170.565	170.154
1600 × 1600	222.399	220.579	221.788	222.024	222.322	226.443
1800 × 1800	290.897	291.087	291.301	291.407	294.728	296.894
2000 × 2000	359.388	360.973	360.214	367.307	356.216	344.524

Table 6: Complete data set for varying landscape size.

% Land	**Run Time(s)**					
0%	26.948	28.551	27.910	29.626	28.183	27.691
10%	35.243	35.205	35.047	35.283	35.396	35.147
20%	43.009	43.200	43.356	42.989	43.240	43.471
30%	51.306	51.047	50.895	50.968	51.930	51.395
40%	58.890	58.969	58.850	58.850	59.081	58.963
50%	68.514	68.210	67.446	70.601	68.172	67.668
60%	78.067	76.551	76.844	75.880	78.069	75.977
70%	85.069	83.067	84.158	83.562	84.934	83.808
80%	91.935	92.109	91.635	91.087	95.651	92.746
90%	100.567	105.926	103.706	102.723	102.573	100.889
100%	109.937	109.308	109.279	110.818	109.236	109.187

Table 7: Complete data set for varying ratio of land to water.

B Statistical Equations and Calculations

B.1 Statistical Calculations

Our data sets were small due to the fact jobs had to be submitted individually to stop them slowing each other down. Because of this, we choose to use the **corrected sample standard deviation** given by:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} \tag{1}$$

which compensates for the bias in standard deviation which occurs for a small data set. The **standard error** was then estimated in the usual way using:

$$\sigma_x \approx \frac{s}{\sqrt{N}} \tag{2}$$

B.2 Parameter Calculation

Given that we have some relationship between variables of the form:

$$y = ax^k \tag{3}$$

then taking a general log of any base we get:

$$\log y = k \log x + \log a \tag{4}$$

which is just the equation for a straight line between the variables $Y = \log y$ and $X = \log x$ with gradient k and y-intecept $C = \log a$. Hence if we can plot the logarithms of the values of x and y we can read of the exponent k as the gradient and calculate $a = b^C$ where b is the base of the logarithm.

C Input Parameters

These are the input parameters in the "input_parameters.txt" used for all simulation tests, except in the case were the performance of the program was tested by varying the number of time steps between output of the plain .ppm file:

- Birthrate of prey 0.08

- Predation rate at which predators eat prey 0.04
- Birth rate of predators per one prey eaten 0.02
- Predator mortality rate 0.06
- Diffusion rate of prey 0.2
- Diffusion rate of predators 0.2
- Size of time step 0.4
- Number of time steps between output of plain .ppm file 10