|epcc|


# Programming Skills
# Performance Testing

B087928

November 30, 2015

# 1 Introduction

When we develop correctly working code we are then usually interested in performance. Knowledge of software used and good coding practise may help us to develop code that works efficiently but we wish to test that this is the case.

If we want to be confident that we have developed efficient code then it is necessary to conduct performance tests. In this report we do this for population dynamics code that we developed in a previous coursework.

Our aim is to see how well our code works; this shall involve determining where the code spends a lot of its time and why. This will give us an insight to whether or not there is a lot of time wasted in areas where a lot of calculation is not required. We may find that our coding technique was inappropriate for the problem.

Throughout our analysis we have only timed the main iterative loop as all expensive computation takes place in this region and in particular, as we shall see from the profiling section, in the update simulation and update animal functions. Morar was used to produce all results for this report.

# 2 Performance tests and Analysis

## 2.1 Compiler Flags

Turning on optimization flags makes the compiler attempt to improve the code performance at the expense of compilation time and possibly the ability to debug the program.

By default the gcc compiler (which was used here) has the flag -O0 which aims to minimise compilation time. Each compiler flag has its own advantages and disadvantages.

In this section we shall examine the effect of different compiler optimisation flags on the runtime to find which one is most appropriate for our program. We used a bash script to compile and run the program with a different compiler flag each time. We ran the program ten times for each flag and took the average as the expectation value for the runtime of the program with that flag. The standard deviation of these values was used as the error. The results may be seen in table 1.

Table 1: Runtime for different compiler flags.

| Compiler Flag | Runtime (s) |
|---|---|
| -O0 (Default) | $14.75 \pm 5.54$ |
| -O | $10.96 \pm 3.81$ |
| -O1 | $9.97 \pm 1.74$ |
| -O2 | $10.31 \pm 3.21$ |
| -O3 | $9.72 \pm 1.76$ |
| -Ofast | $6.20 \pm 1.30$ |
| -Os | $8.43 \pm 2.72$ |
| -Og | $11.05 \pm 4.63$ |

We expect -O, -O1, -O2, -O3, -Ofast to each be faster than the last since they each contain most of the optimisations of the previous flag plus something extra. -Ofast can produce a highly sequentially efficient executable. -Os aims to reduce the size of the code. -Og only allows optimisations that do not interfere with debugging, so we don't expect it to be the most efficient but is useful if we are compiling the code a lot while we are debugging.

We ran these simulations on a $100 \times 100$ landscape, 100% of which was land. From the results we determined that -Ofast was the best optimisation flag for the problem. This flag is used throughout the rest of the performance testing.

## 2.2 Profiling

To identify and quantify the main source of overhead we included -pg compilation flag. Running with this flag allowed us to examine the profile of the program using pgprof to see where the code was spending most of its time.

We analysed landscapes that were 100% land and of size $100 \times 100$, the results may be seen in table 2. Approximately 99% of time is spent in these three functions alone. Thus we know this is where we need to look for performance issues.

Table 2: Percentage time spent in functions. 100% Land

| Functions | Percentage of Time |
|---|---|
| update_simulation | 48.69% |
| update_animal | 41.65% |
| get_neighbour_count | 7.51% |

In table 3 we have analysis from a 10% land configuration. Comparing tables 2 and 3 we can see that if we have a much lower proportion of land we spend most of our time in the update_simulation function. This is because we do not need to call update_animal for all water grids throughout the landscape.

Table 3: Percentage time spend in functions. 10% Land

| Functions | Percentage of Time |
|---|---|
| update_simulation | 86.70% |
| update_animal | 13.24% |

The runtime for the 100% and 10% land simulations were $6.41$s and $5.09$s respectively. This difference in runtime can be predicted to be due to the time spent in the update_animal and get_neighbour_count functions.

## 2.3 Proportion of Land and Water

In figure 1 we have plotted runtime as a function of percentage land for a $100\times100$ landscape. We computed the time at each percentage 10 times and used the average as the expectation value and the standard deviation as the error.

It is obvious that if we have no land, only water, then there is nothing significant to calculate. However, when we run our program it must loop through the entire landscape at every timestep regardless of whether or not there has been a change. This explains why we have significant compute time even when 0% of our landscape is land. We can see, as we would expect, that a higher percent of land incurs a higher runtime. But it is notable that the performance of the program does not depend very heavily on the amount of land- the extra compute time here is predominantly due to the time spent in update_animal as was found when we examined the profile.

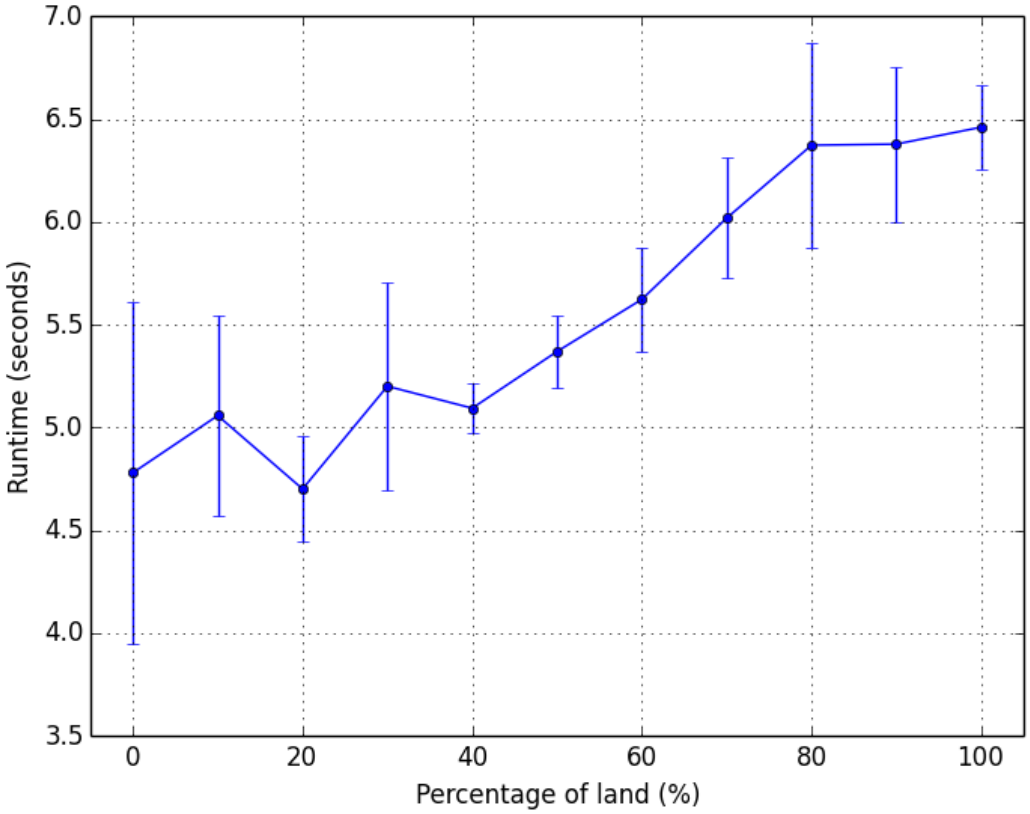


Figure 1: Runtime against percentage of landscape land.

It is obvious from this analysis that the choice to do a sweep of the entire landscape to perform an update is one that effects performance significantly. For simulations with water we are spending a large percentage of compute time in areas where there is nothing to calculate.

## 2.4 Landscape Size

Finally we'd like to examine the effect of landscape size on computation time. For these simulations we have used landscapes with 100% land.

We can see in figure 2 that runtime depends heavily on landscape size. Runtime is directly proportional to the area of the landscape. This implies that the program is spending almost all its time working through the grids one by one. Therefore there is very little overhead which is consistent with our results from profiling.

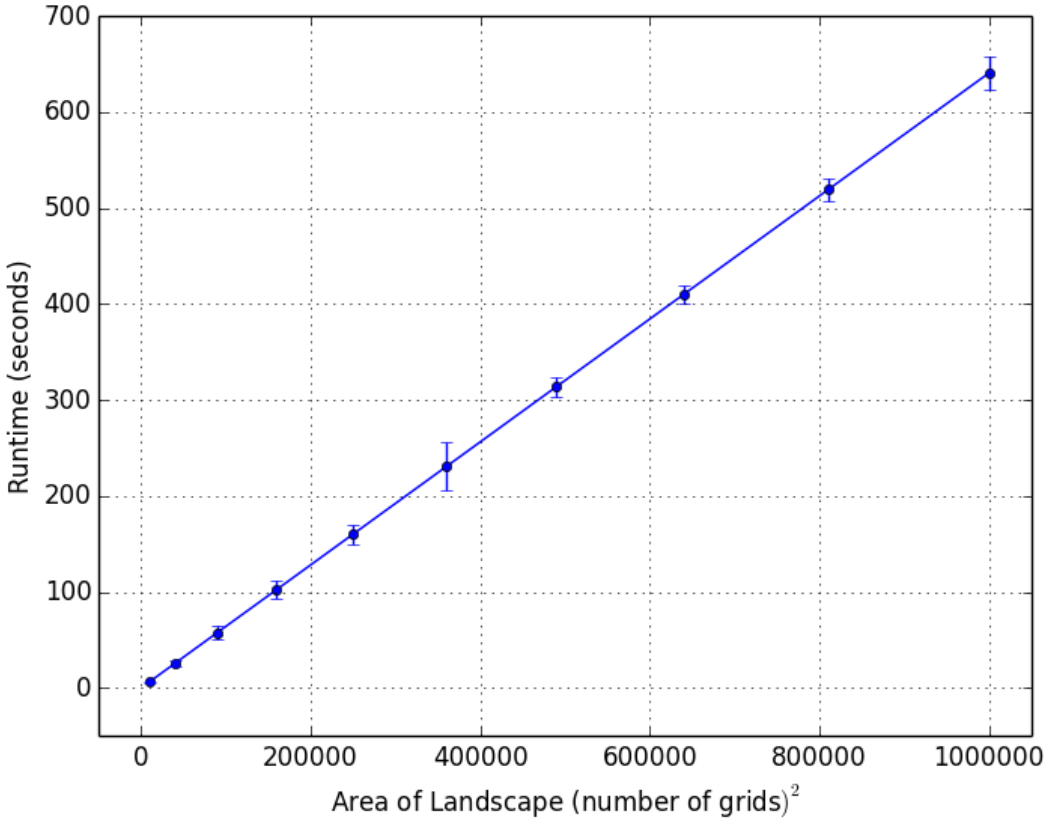


Figure 2: Runtime against size of landscape.

# 3 Conclusions

The program spends most of its time in update simulation where there are many loops and conditional statements that take time to work through. A consequence of this is that grid points where there is nothing to calculate do not cost much less than grid points where we need to calculate the population change. Our approach to solving the problem by visiting every grid point each iteration causes there to be very little overhead but also means that we do a lot of unnecessary calculations due to the program examining grids that contain water at every iteration.

Further code development and tests should involve producing a program with a higher overhead that only needs to loop over land grids. Only then may it be determined which approach gives best results for given landscapes. It is expected that our solution is the most appropriate for landscapes that are mostly land. This is because in this case it is necessary to update every

grid point each iteration. However, with this suggested alteration, we'd expect to see a larger difference in performance between majority water landscapes and majority land landscapes than that seen in figure 1.

Performance analysis can be very important in determining the correctness of our approach to problems and to highlight possible improvements or alternative algorithms. We conclude that the most important elements and the focus of our analysis should be determining on what tasks the program spends most of its time; if this is where we expect; if the distribution of time is sensible or should the program be able to do some tasks faster. Ways that we did this in this report were profiling and examining the effect of different input data on runtime and determining the cause.

When we are happy with our program architecture we can see what compiler optimisations give us the best resulting performance.