Programming Skills
Hare and Puma Population Dynamics
Group Report

Cian Booth, **Pádraig Ó'Conbhuí** and **Ruairi Short**

November 9, 2012

1 Introduction

A pair of coupled PDEs can be used to model a predator prey model, like that of hares and pumas on an island. An example set of PDEs is

$$\frac{\partial H}{\partial t} = rH - aHP + k\,\nabla^2 H$$
$$\frac{\partial P}{\partial t} = bHP - mP + l\,\nabla^2 P$$

Where H and P are 2D fields representing the densities of hares and pumas at a given point in space, t is time, r is the rate of birth of hares, a is the rate of predation of hares by pumas, k is the diffusion coefficient of the hares, b is the rate of birth of pumas (dependant on the number of hares to feed their young), m is the rate of death of pumas (by old age) and l is the diffusion coefficient of the pumas.

This set of PDEs can be discretized to be solved numerically. This can be done by defining a grid of points where the fields are defined and the values of the fields at a time $t + \Delta t$ can be found using the following update routines

$$H_{i,j}^{t+\Delta t} = H_{i,j}^t + \Delta t(rH_{i,j}^t - aH_{i,j}^t P_{i,j}^t + k((H_{i-1,j}^t + H_{i+1,j}^t + H_{i,j-1}^t + H_{i,j+1}^t) - N_{i,j}H_{i,j}^t))$$
$$P_{i,j}^{t+\Delta t} = P_{i,j}^t + \Delta t(bH_{i,j}^t P_{i,j}^t - mP_{i,j}^t + l((P_{i-1,j}^t + P_{i+1,j}^t + P_{i,j-1}^t + P_{i,j+1}^t) - N_{i,j}P_{i,j}^t))$$

A program was designed and written to do just this, where a user could specify the value of each variable, and the program would output some data on the fields at particular intervals.

A group of three was formed to undertake this task. Given the relative simplicity of the problem, and the group members' maths and programming backgrounds, it was felt that any more than three members would spread the work too thinly.

2 Planning

Firstly, the group listed the requirements given in the specification. They are as follows;

- Program should be able to take in an input file, the form of which is given in an example file. This file could contain up to 2000×2000 elements.
- User should be able to input the parameters that affect the evolution of the system, r, a, b,... etc.
- Program should be able to output a plain PPM file that represents the current state of the system.
- User should be able to input the output frequency T, which governs how often a plain PPM is written.
- The mean values of hares and pumas over the grid should be outputted at regular intervals.
- Programming Skills techniques should be used.

As the group consisted of three members, it seemed obvious to attempt to initially split the problem up into three sections. Persons who completed their section faster than others, could go on to optimise their section of code, or otherwise further enhance the program overall. Equally, if full testing was not implemented, more tests could be added at this point.

The three sections were decided to be "Input", "Update" and "Output".

2.1 Input

The input section would mainly entail the reading of the input file. However, this naturally implies the creation of a system to store the contents of the input file. Thus, the input section also contains the storage and access of the 'board' as it is read and written to.

Naturally, the input section also holds domain over the user input options, which would take the form of command line options. Again, this implies the creation of error messages on incorrect usage of the program.

Extra time would be given to increasing the efficiency of the storage and reading of the 'board'.

2.2 Update

The update section consisted mainly of implementing the problem's algorithm. It was important that this section either returns a board, or edits the board in place.

Optimisation and speed-ups would be especially important in this section. Thus, this section also commands the use of parallelism and other efficiency based techniques, especially those that apply to the algorithm.

Extra time would be devoted to adding extra tests, of which there can never be too many for such a problem.

2.3 Output

This section is devoted to the requirements pertaining to outputs. This of course, mainly means the outputting of the plain PPM file, and determining how it would represent the system.

Equally, statistics about the 'board' needed to be outputted, and as such this section also has governance over creating and showing the statistics to the user.

Extra time would be applied to adding a GUI to the system, and all the things such a thing requires.

3 Assignment of Tasks

Once a plan had been made it was necessary for the tasks associated with the program to be divided up among the group. The first task was to decide on a build tool to use. `Automake` was settled upon and Pádraig took charge of implementing this as he had previous experience with its use. Next a method of version control was decided upon. `Git` was the chosen tool and an online repository was managed by Cian on the website `github` Since main program naturally divided itself into three parts each team member was then polled and was assigned to the task they would most like to do. The split was thus as follows;

- Input - Pádraig Ó'Conbhuí
- Update Operation - Ruairi Short
- Output - Cian Booth

Pádraig did the input part of the program as he had the most experience with `C++` features such as memory allocation that was necessary to implement this part correctly and efficiently.

Ruairi was in charge of the update operation. This part involved the most maths and having a background in physics meant that this part was ideal for him to work on. It was also necessary to have tests that would ensure that the update was working correctly. In order to this the equations needed to be solved analytically for certain situations and tests created to reproduce these situations.

Cian had a desire to implement the output part of the code and had previous experience with writing GUIs in `C++`.

Each group member was also responsible for testing their own part of the code. This was since they would have the best understanding of the module they had written and could test it most effectively. It also meant that it was possible to maintain a working version of the code throughout the entire project.

Once the code program was running to a sufficient standard the group met again to look at what further tasks could be done. The conclusion was that the remaining tasks were attempting to improve the performance of the program overall, implementing a graphical user interface(gui) and implementing tests to check overall functionality of the main program. It was decided that some time should be spent trying to improve performance. It was speculated that the most likely user of CPU time would be the update function so Ruairi took charge of performance analysis and optimisation. This involved profiling the code to find out if the speculation had been correct and acting on the data from the profiling. Cian did the work on creating a gui using `ncurses` so that the output from the program could be better understood by the user as the PPM output was not immediately available while the program was running. Pádraig did further work testing the input part of the code to ensure that it was working efficiently and correctly.

4 Programming Language

As a group, it was decided that the programming language C++ was the right language to use for this project. There were several reasons for this. The main reason was that each member of the group was well versed in the use of C++. Familiarity ensures that the production of code is fast and efficient. Furthermore, using a programming language that the group was accustomed to, meant that pertinent libraries could be used.

Another important reason to use C++ is that, compared to the likes of Fortran, input and output operations are considerably easier, and better implemented. In a project where reading and writing files is a requirement, it is obviously beneficial to use a programming language that is strong in this area.

Equally, C++ has many libraries (courtesy of C compatibility) and compilers that ensure optimised code, that can run at the highest efficiency, unlike languages like Java. In particular, the library OpenMP is particularly useful for improving efficiency.

C++ was chosen over C because of the inherent memory management within. This is useful in a project which could potentially be using very large arrays.

Finally, again courtesy of the C compatibility, C++ has access to many different varieties of GUI libraries. This include, but are not limited to, the very commonly used wxWidgets (seen in programs like Audacity and Dropbox), the popular library Qt (used by programs like Skype and Mathematica) and the terminal based ncurses, used by the likes of Screen and cmus. Thus it is a good choice, when building a GUI comes about.

5 Revision Control

The group decided to use Git as the revision control system for this project. This was selected since it was possible for each group member to maintain a local repository and a master one was maintained online at github.com. This meant that as long as internet access was available everyone could see the latest updates and additions to the project but if there was not they could still work on their own code.

5.1 Other Tools

The other tool considered for use was SVN. This is also a widely used revision control package. It has the disadvantage that the user must be online with respect to the master repository. Since all the group members already had Git installed on their computers and membership of the online repository github it was much simpler to use this as the revision control system.

5.2 Features

Git allowed each member to maintain their own local repository separate from the master. This meant that testing, updates and bug fixes could be done by each member without affecting the other group members' work. By creating a separate branch on ones machine it was easy to keep track of the changes you were making without having to merge conflicts with other users. Another feature of Git is that it is quite

good at resolving conflicts and merging branches. As there were times when two or more people might try and edit the same file this aspect was used quite often. With this revision control system it was easy to keep track of the development of the program and everyone could see immediately when new files had been created and the commit messages were useful in understanding what changes had occurred and for keeping track of information such as performance data. Also of use as a feature in `Git` was the fact that one could go back a version in the code if drastic mistakes were made! It was found that it is important to keep the .gitignore file up-to-date as compiled and created files often change when code is being changed or updated. `Git` cannot find the differences between binaries and since `Automake` was being used it was very easy to generate these files. There were some disadvantages with using `Git` in combination with `Automake` however as if another group member reran the `autoreconf` command then all other users would have to run it and this took some time, which could be frustrating. A pre-commit script was written in an attempt to reduce the effects of this that took the names of any executables created by the makefile and added them to the .gitignore file.

Overall `Git` was found to be a very useful tool when it came to revision control. It was well suited to this type of project and would be used again.

6 Build Tools

For this project, the `Autotools` suite was chosen. This is a free set of tools for generating portable programs for many Unix-like systems.

6.1 Other Tools

The other main competitor for use in this project was plain `make`. While a plain `Makefile` would be easier for a small, simple project, we felt that `Automake` was a more robust build system, better suited to a project such as this one. Two others that came up were `cmake`, when another group mentioned it was their build tool of choice, and `qmake`, when `Qt` was considered for use in building a GUI. However, much time and effort had already been invested in setting up an `Automake` build environment, and the other build systems offered no particular advantages worth switching environments for.

6.2 Features

The `Autotools` suite is a widely used set of tools for generating portable makefiles and configuration scripts. Since the tools are widely used, many users are already familiar with the procedure for compiling a project using these tools. This is the familiar "`./configure && make && make install`" combination. The configuration script can be used to find or explicitly specify the locations of certain libraries on a system, along with specifying install paths for the compiled program. It as also used to ensure the appropriate build tools, libraries, and library headers exist on a system, and give some descriptive error messages when these can't be found. The alternative is for a user to attempt to decrypt some mysterious error messages when a program either outputs errors at compile time, or does compile and outputs an error at run time.

The `Automake` tool specifies a simple way of defining programs to be compiled, the dependencies of these programs, and the flags to be passed to them. The tool makes it simple to specify these things on a global level, or a per program level. It also define some standard `Makefile` targets, such as `clean`, `distclean`, `check` and `dist` which have some standard functions. These are supplied with no extra configuration, whereas with a plain `Makefile`, these would have to be hand-coded. `Automake` also reduces the need for "clever" `Makefiles`, which are almost always a nightmare for anyone other than the original author to maintain (assuming that they themselves understand it).

Also provided is a rudimentary test environment. Programs that might be compiled for a test run are added to a list, separate from the ones that are compiled for installation, and the programs to be run during a test run are added to another list. For this project, it turned out that just one test environment wasn't quite enough, but since `Automake` provides the means for including pure `make` targets, it was trivial to add new targets for compiling and running specific sets of tests. This meant the main tests could be run by simply running "make check", the performance tests coule be run by running "make performancetest" and the gui tests could be run by running "make guitest". This makes testing slightly less painful for the programmer.

The main problem with the `Autotools` suite is that is can have quite a steep learning curve to learn all the gory details involved in it. However, in most cases, these can simply be ignored. This is because of one of the main strengths of the suite: its widespread use. That is, there are plenty of tutorials for using the tools and plenty of questions and answers available on the internet. Another problem is that the configuration script can take a while to run, and needs to be run again if any changes are made to any of the files used to generate the `configure` script or any of the `Makefiles`. This can be quite tedious.

7 Testing

Three sets of tests were set up for this project:

- Correctness tests
- Performance tests
- Output tests

7.1 Correctness Tests

The approach taken for correctness testing was to write one program to do each unit test. Upon the first error in the test, the program would exit early and return a value of 1. The idea behind this is that it would eliminate the overhead of learning a whole unit testing framework that might return the number of tests passed and the number failed. We felt this would be unnecessary, since if only one test fails, it's not really necessary to see how many others passed and how many failed, as the program is broken, regardless of the outcome of other tests. If more details are required, a programmer can simply comment out the offending test to see which others pass and fail.

Finally, an integration test was built in the form of a bash script that passes several input files into the program and ensures it returns the correct output in some special cases.

- Given the equations

$$\frac{\partial H}{\partial t} = -aHP$$
$$\frac{\partial P}{\partial t} = bHP$$

  The following equality holds

$$\frac{\partial H}{\partial t} = -\frac{a}{b}\frac{\partial P}{\partial t}$$

  This should test that the two PDEs are properly coupled.

- Given an input with the hare or puma densities in some squares set to zero, when the diffusion terms are set to zero, these densities should remain at zero.

- When only the hare birth term, r, and puma death term, m, are nonzero, the hare and puma populations should follow the equations

$$H(t) = H_0 e^{rt}$$
$$P(t) = P_0 e^{-mt}$$

- When only diffusion terms are nonzero, both hare densities and puma densities should reach an equilibrium distribution.

- When an equilibrium distribution is given as input to the program, given the same input values as the previous test, the program should output the exact same board (or reasonably close to it, as.

- When a checkerboard distribution is given and only diffusion terms are nonzero, zero density tiles should not stay at zero.

7.2 Performance Tests

The main set of performance tests that were run was on the update_animals function. This was due to the fact that profiling indicated that at first it was the major user of CPU time. In order to see more easily how small changes were impacting the performance of the function a performance testing piece of code was written. It had the objective of running on a big enough board that the results were appropriate for real simulations and over enough iterations so that any overheads associated with calling the function would show up. To this end the test was written for a 1000x1000 board and 50 iterations.

As output this test produced the time taken and the theoretical flops for the operation. The flops were computed by considering only the operations that one reads as being necessary in order to compute the actual update equations. Any other operations are implementation dependent and thus not included. It was then possible to keep track of the improvements easily by adding the timing and flops in the commit messages when committing to the online `Git` repository.

7.3 Output Tests

Output testing for writing `.ppm`s was done in a comprehensive fashion. It is simple to test output files, but inputting a known file, and immediately outputting that file. If the values in the output file correspond to the input, then the system works.

Several cases were developed to test the output;

First Line Validity A plain PPM file will be invalid if the first line does not contain the correct information. It requires the text "P3", to indicate that it is a plain PPM. It then needs the width and height of the data to be drawn. Finally, it needs to know the maximum value of the data within (which is normally 255). This test ensures that a sample input file of size 5×5 creates an output file with the first line reading "P3 5 5 255"

Number of Lines Plain PPM files also require that the number of lines in the file is correct. The number of lines in the document should correspond to the second digit indicated on the first line plus 1 (to include the first line). Again, an input of size 5×5 was used to check this.

Note that the output of this program actually introduces extra spaces between lines, to increase ASCII readability of the file.

Number of Elements A final check of the validity of the plain PPM output file, that the number of pixels represented is the same as the number of input elements. The program outputs the data as three integers, with spaces separating them. Between each data point, is a tab space. This can be used to count the number of pixels drawn in each line, which is the number of elements in the output grid. In combination with the above test, this implies that the output file is indeed laid out correctly.

No Hares, No Pumas, Only Land In this test, the system is set into a state where there are no rabbits, hares, and there is only land. In this case, all the pixel values in the output file should be "0 0 0".

No Hares, No Pumas, Only Water In this test, the system is set as above, except with the entire region being made of water. If this is the case, the pixel values should all be "0 0 255", indicating no animals and only water.

Constant Hares, Constant Pumas In this test, as the above test, except there are now non-zero numbers of hares and pumas. Again, the expected result is that the pixel values are "0 0 255", ignoring the data of the hare and puma.

This test should have no particular use, as water tiles cannot, in the program, be populated by any animals. However, it does ensure that the water system is printing correctly.

Constant Hares, No Pumas, Only Land Here, the number of hares is the same in all grid points, 1. The region is entirely made of land. Therefore the pixel count of each element should be "0 1 0".

This test ensures that hares are being added to the output properly.

No Hares, Constant Pumas, Only Land As above, but with no hares and every puma set to a value of 2. The pixel count is expected to be ”2 0 0” for each element.

This test ensures that pumas are being added to the output properly.

Constant Hares, Constant Pumas, Only Land Now there is a constant number of hares, 1, and a constant number of pumas, 2, in each element. This leads to a pixel count of ”2 1 0”.

This test makes sure that the hares and pumas are not somehow interfering with each other's output.

Linear Hares, No Pumas, Only Land Now each element has a number of hares equal to the sum of it's column and row minus 1, if they are numbered from 1 to 5. Thus, the first element in the first row is expected to have a pixel value of ”1 0 0”, the second ”2 0 0” etc., all the way to the last element on the last row having a pixel value of ”9 0 0”.

This test ensures that the number of being outputted is independent of other values of hare.

No Hares, Linear Pumas, Only Land As above, but with pumas, and the rows and columns are numbered from 5 to 1.

Non-Integer Hares, No Pumas, Only Land This test is as the Linear Hares, No Pumas, Only Land test, except that each element is now multiplied by 1.1. This means that a full range of decimal places is tested. It is expected the output will round the value down, giving the same results as the aforementioned test.

This test ensures that the plain PPM does not contain any non-integer values.

No Hares, Non-Integer Pumas, Only Land As above but for pumas.

Sample Input The sample input is a much larger file type, 24×24 elements. It contains regions with water, regions with no hares or pumas,with just hares, with just pumas, and regions with both species. This stress tests the system to make sure that the previous tests pass in unison.

Sample Input Xstretch This is the same as the above system, except that now the board is rectangular, having been stretched by one in the x direction to 25×24.

This ensures that the output is not reliant on symmetric systems.

Sample Input Ystretch As above, except that it has been stretched in the y direction, to become 24×25.

This test ensures that there is not some special dependence on either symmetry of the board or the x direction being stretched.

8 Debugging

For the most part debugging was done by the group member responsible for the part of the code where the code was found. This was done as they were the most familiar with it and thus were the most likely to know what the problem was. There

were a number of occasions where the group did work together to try and resolve bugs that were found. It was ensured that whenever a bug was found, a test would be written to catch it so that in future it would be noticed if it re-occurred. Ordinarily the causes of the bugs were found using print statements as most bugs found during the project were quite small and the problem was almost always immediately obvious. It is possible that some of the bugs would have been found more easily if debugging tools were used but due to the overhead associated with learning the use of most tools it was not worthwhile for this project.

The `Memcheck` tool included with `Valgrind` was used however in order to check that the memory was being allocated and deallocated correctly and that no memory leaks were occurring. This was found to be this case so no further action was required.

9 Performance Tests and Analysis

In order to decide what parts of the code needed to be optimised it was necessary to run some performance tests. As a precursor to this the main code was profiled so that it was possible to decide what parts of the code needed work. The profiling program `Valgrind` and the tool `Callgrind` associated with it was used. This showed that one of the large users of CPU time was the update_animals function as would be expected as this is the main computational part of the code. Once it was known what that this part of the code needed testing a specific performance test was written to test this function as outlined in the tests part of this report. Since a comprehensive test suite had already been written the code could be changed quite aggressively in order to improve performance.

The first idea was to investigate compiler flags. `G++` was the compiler used for this project so the compiler options `-O1`, `-O2` and `-O3` were tested in the hope that they might improve performance. It was seen that as optimisation levels were increased there was increase in the performance of the code as could be expected. There was however a negligible difference seen between the `-O2` and `-O3` options. Due to the fact that `-O3` is more aggressive when it comes to changing the execution of code it was decided to use -O2 as the compiler optimisation option.

Next the memory accesses during the update were considered. It was noted that it was necessary to compute the sums over the number of adjacent land tiles and numbers of hares and pumas in these tiles. These sums were brought together before the main update operation with the intention of reducing the number of cache misses and keep memory accesses close together. There was a performance improvement seen by doing this.

Since the update could not be done in place a new board was declared in the `update_animals` function. This cause an overhead to be associated with calling the function. In order to reduce this overhead the board was declared as static so the declaration was only needed once. As a result of this the execution time was almost halved.

It was though that for large board sizes that parallelizing the update would be beneficial. OpenMP directives were added in order to make this possible. Due to the way that the update is applied there were no race conditions on any variables and this made the implementation relatively easy. These allowed for the program to be easily switched in and out of parallel if the library was not available on some machine for example, and it allowed for a parallel and serial version of the code to be main-

tained together. This parallelism was then tested on a number of different threads. For small boards it was found that the parallel implementation could be slower due to the overheads associated with creating threads but for larger grids there was a significant improvement in performance. Since it is up to the user to specify the number of threads and the board size it was decided that this was acceptable. For small grids the execution was fast anyway so the user would not notice any impact if they did not choose the best options.

These simple optimisation techniques were seen to be easily implemented but also very effective. The result was when the `Valgrind callgrind` tool was used again to profile the code the I/O part was the main user of CPU time. It was decided at this point that the program ran at a sufficiently fast speed and that the time spent trying to improve the I/O part of the code could be better spent on other improvements to the program.

10 Conclusion

A program that could visualise the population dynamics of hares and pumas was developed. It included two output modes, writing .ppm files, and showing the output with an ncurses GUI. The input was implemented, allowing users to include their own input files, and their own parameters.

The group worked together very well, which was especially down to the good planning done before the actual coding was done. The separation of work went well, as we all finished up roughly within a day of each other. The work was given to the appropriate people, as befitted their egregious talents.

The program was produced using many Programming Skills techniques. A combination of git and github was used for revision control. Automake was used as a build tool. Valgrind was used as a profiling suite. The tests implemented saved a lot of time and bug checking. Debugging was done by hand, as the testing and program design meant that big bugs didn't really appear. The code was optimised using compiler flags, and parallelised using OpenMP.

In conclusion, the project was a success.

11 Further Work

11.1 Input

The program fails to build with the current framework when pgcpp is used as the `C++` compiler. The problem appears, on the surface at least, to be with the boost library. Future work might entail the removal of the boost program_options library in favour of wither a custom option, involving parsing a rigid input file, or perhaps more simply, requiring variables to be set on compile. This may, in fact, not be the problem, as the compilation attempt with pgcpp was only done on the Morar machine. It may be possible to bundle the program_options library with the source of this project, getting a user to compile it with pgcpp along with the rest of the code.

11.2 Update

In the update section there is more work to be done in optimising the code to run faster. However one can almost always say this about a piece of code! A number of possibilities were considered but due to time constraints were not implemented.

For example to reduce further the overhead of copying the board back from the working board back to the board visible from the main program one could simply move the pointer. This could be quite difficult to implement in reality as it is not known exactly how this might behave with the board class but it could reduce the number of operations and this increase performance. Alternatively having the working board initialised in the main program could potentially speed up execution.

There is also more work to do from the point of view of parallelizing the loop. Different loop scheduling options for OpenMP could be experimented with in order to find an optimum. The use of a message passing protocol such as MPI could also be tested as the communications pattern is quite regular and could possibly see good performance.

It is possible that there is more to be done with improving cache accesses and overheads associated with loops by techniques such as loop unrolling as well but it was decided to not implement any of these in order to keep the code readable.

11.3 Output

There is a lot of work that could be done with the output section. The most pressing thing that needs to be done is testing the GUI. Although this is difficult, due to changing terminal size, it could be possible to fix this, as far as the program is concerned, using pre-processing conditionals in the code, forcing the ncurses set values of COLS and ROWS to standard values.

Full user interactivity with the `write_adjustable_ppm` function could also be added, which could be useful for smaller input files.

The ncurses section of the program was also not implemented very efficiently. It spends extended periods of time waiting, time which could be spent calculating upcoming iterations. Extraneous iterations could be saved in a file, and whilst this is potentially a way to use up large amounts of memory, it is worth looking into.

Finally, if a more universally accessible, and less cumbersome windows based GUI could be found, the quality and usability of the program could be greatly enhanced, as the majority of people are very used to using windows based programs.