

An Introduction to High Performance Computing

# Crash-course on Writing Reports

Andrei Poenaru

17 October 2019

# Why?

- This is a report-based coursework, so the overall quality of your report is **very** important
  - You need to present what you did—just submitting the code isn't enough
- Not many occasions to get feedback on your writing so far
  - We see a number of common mistakes repeated every year
  - Start thinking early about how your reader will perceive your text
- This session will touch on a wide range of points
  - Not all may be applicable to each of you
  - Don't do things just because they're mentioned here—there isn't a *single* right way, and you need to combine good writing principles with *your* style

# What you want to avoid

## I. INTRODUCTION

This assignment involved taking the lattice Boltzmann code provided and applying serial optimisations, as well as the OpenMP API to parallelise the code, enabling it to run on all cores within both sockets of a BCP3 node.

Given that OpenMP is focused on shared memory paradigms, whilst OpenMPI (the standard used in the previous coursework assignment) is centred around a message passing paradigm, the strategy used within OpenMP is vastly different; there will be less focus on the sending and receiving of messages with the halo exchanges, and more focus on running multiple blocks of code in parallel, with commands such as `pragma`.

This report will include detailed descriptions of the serial optimisations I chose to make, as well as my method of parallelism, and analysis into performance. Any runtimes or speed-ups referenced will be for the 128x128 image, unless stated otherwise.

## II. SERIAL OPTIMISATIONS

My first step involved compiling and running the program - the average runtime was 58.8 seconds.

I then immediately optimised the compiler. I decided to implement both `gcc` and `icc`, and compare by using the equivalent optimisation flags on both compilers. Interestingly, even at the first stage, using `icc` was beneficial; `gcc` produced a speed-up of 1.8x to 32.5 seconds, whilst `icc` produced a speed-up of almost 2x to 30 seconds - I therefore opted for `icc` in this assignment, using standard compiler flags such as `-Ofast` for statement optimisation and `-xHost` to tailor the program to the processor.

An initial traversal through the code led me to an obvious bottleneck - the repetition of the two `for` loops through `ij` and `ji`, operating on a large number of cells. It seemed logical to combine the operations within the functions `rebound`, `accelerate`, `flow`, `collide` and `propagate` into a single function via loop fusion, as they contained almost identical parameters, and operated on the same data. After some experimentation, I managed to simplify this into a single function, named `calculate`, which encompassed all of the behaviour of

the above functions, meaning I could safely discard the `timestep` function, which acted as an intermediate step between initialisation and the functions themselves. This greatly reduced cell access (by almost a factor of 3), and measured a speed-up of around 1.6x.

An additional complication involved with the program was the unnecessary data accesses within `tmp_cells` and `cells`. The idea behind the pointers is to mimic the behaviour of a double buffer, but the method used is incredibly inefficient - once the data is written from `tmp_cells` from `cells`, it is copied back again, leading to twice the number of data accesses. It would be beneficial therefore to 'update' `tmp_cells` as cells to avoid this extra data access - this produced a 1.5x speedup.

Ordinarily, when a calculation is trivial and made multiple times (such as the indexing in the stencil assignment), the compiler optimises it automatically. However, the compiler was unable to carry out the same simplification in this instance due to the increased complexity, and therefore storing pre-calculated values such as collision constants actually improved runtime, despite the additional requirement to fetch the value from memory. This step produced a speed-up of around 1.2x.

Some other minor optimisations included replacing division with multiplication where possible (as multiplication can be completed in constant time  $O(1)$ , whereas division can be as large as  $O(N^2)$  in cases such as long division), switching constants such as 1.1 to 1.1f to avoid the need for conversion, and swapping `i` and `j` to reduce 'jumping' between cells. After all of these optimisations were applied, the runtime settled at around 13 seconds. I was unable to implement the vectorisation fully, but comfortably surpassed the 22 seconds needed for the serial optimisation.

Size	128x128	256x256	1024x1024
Full-Params (s)	22.0	170.0	720.0
Optimised (s)	13.1	106.5	444.2

Figure 1: Change in runtime after optimisation.

## I. INTRODUCTION

This assignment involved taking the lattice Boltzmann code provided and applying serial optimisations, as well as the OpenMP API to parallelise the code, enabling it to run on all cores within both sockets of a BCP3 node.

Given that OpenMP is focused on shared memory paradigms, whilst OpenMPI (the standard used in the previous coursework assignment) is centred around a message passing paradigm, the strategy used within OpenMP is vastly different; there will be less focus on the sending and receiving of messages with the halo exchanges, and more focus on running multiple blocks of code in parallel, with commands such as `pragma`.

This report will include detailed descriptions of the serial optimisations I chose to make, as well as my method of parallelism, and analysis into performance. Any runtimes or speed-ups referenced will be for the 128x128 image, unless stated otherwise.

## II. SERIAL OPTIMISATIONS

My first step involved compiling and running the program - the average runtime was 58.8 seconds.

I then immediately optimised the compiler. I decided to implement both `gcc` and `icc`, and compare by using the equivalent optimisation flags on both compilers. Interestingly, even at the first stage, using `icc` was beneficial; `gcc` produced a speed-up of 1.8x to 32.5 seconds, whilst `icc` produced a speed-up of almost 2x to 30 seconds - I therefore opted for `icc` in this assignment, using standard compiler flags such as `-Ofast` for statement optimisation and `-xHost` to tailor the program to the processor.

An initial traversal through the code led me to an obvious bottleneck - the repetition of the two `for` loops through `ij` and `ji`, operating on a large number of cells. It seemed logical to combine the operations within the functions `rebound`, `accelerate`, `flow`, `collide` and `propagate` into a single function via loop fusion, as they contained almost identical parameters, and operated on the same data. After some experimentation, I managed to simplify this into a single function, named `calculate`, which encompassed all of the behaviour of

the above functions, meaning I could safely discard the `timestep` function, which acted as an intermediate step between initialisation and the functions themselves. This greatly reduced cell access (by almost a factor of 3), and measured a speed-up of around 1.6x.

An additional complication involved with the program was the unnecessary data accesses within `tmp_cells` and `cells`. The idea behind the pointers is to mimic the behaviour of a double buffer, but the method used is incredibly inefficient - once the data is written from `tmp_cells` from `cells`, it is copied back again, leading to twice the number of data accesses. It would be beneficial therefore to 'update' `tmp_cells` as cells to avoid this extra data access - this produced a 1.5x speedup.

Ordinarily, when a calculation is trivial and made multiple times (such as the indexing in the stencil assignment), the compiler optimises it automatically. However, the compiler was unable to carry out the same simplification in this instance due to the increased complexity, and therefore storing pre-calculated values such as collision constants actually improved runtime, despite the additional requirement to fetch the value from memory. This step produced a speed-up of around 1.2x.

Some other minor optimisations included replacing division with multiplication where possible (as multiplication can be completed in constant time  $O(1)$ , whereas division can be as large as  $O(N^2)$  in cases such as long division), switching constants such as 1.1 to 1.1f to avoid the need for conversion, and swapping `i` and `j` to reduce 'jumping' between cells. After all of these optimisations were applied, the runtime settled at around 13 seconds. I was unable to implement the vectorisation fully, but comfortably surpassed the 22 seconds needed for the serial optimisation.

Size	128x128	256x256	1024x1024
Full-Params (s)	22.0	170.0	720.0
Optimised (s)	13.1	106.5	444.2

Figure 1: Change in runtime after optimisation.

What about the non-square problem?!

# How?

- Formatting
- Structure and layout
- Content
  - Story
  - Language and style
  - Present findings
  - *Explain* findings
- Proofread

# Conventions

- Black text contains general explanations and suggestions
- **Dark red text** shows examples of **bad** writing
  - Avoid similar issues in your own report
  - Emphasis is mine, highlighting the problems
- **Blue text** shows examples of **good** writing
  - Use similar strategies to improve your own writing

# Formatting – why?

- You are not marked on the aesthetics of your report *per se*
  - But good formatting principles have been developed to increase clarity, so following them can only work in your favour
- When you write—in general—you are trying to convince your reader of your knowledge on the topic
  - They are more likely to take you seriously if they can see you've made an effort to write easy-to-read text
  - They are more likely to believe you if you demonstrate you have paid attention to *other* good pieces of writing, e.g. edited articles or research papers



# Formatting (1)

- Avoid non-standard fonts
  - **Comic Sans** is **not** a sensible choice
  - Monospace fonts are **not** sensible choices
  - Times New Roman is not great, not terrible<sup>3.5 R</sup>
  - Arial is a good sans-serif choice
  - Georgia is a good serif choice
- Pick a sensible font size
  - Try to stay between 10 and 12 pt (but references can be smaller)
    - Any lower and text can become hard to read, any higher and it starts looking as if you're trying to cover up a lack of content
  - Stay below 1.5X line spacing
  - If unsure, print the report and make sure it's comfortable to read

# Formatting (2)

- Use alternative font styles where there may be ambiguity
  - Use *italics* for emphasis
    - But don't overuse it, because then it becomes distracting
    - **Bold** is a *different* kind of emphasis—only use it if you're confident you understand the subtle difference
  - Use a monospace font for inline code
    - It makes it clear where the fragment of code starts and where it ends
- Hyphens and (the several types of) dashes are different: -, —, —  
All versions of the application—regardless of whether inter-procedural optimisation was enabled—ran in 325–330 seconds
  - If you don't know which one is right, rephrase and avoid using them altogether



# Formatting (3)

- If you use “typographic quotes”, make sure you use them correctly
  - It’s better to use “plain quotes” than to “get it wrong”  
The profiler ‘gprof’ was used
- But be careful when quoting terms: it may imply a different meaning
  - Please send me your “notes” might suggest I don’t think they are doing a very good job at writing notes...
  - If you’re simply introducing terms, use italics instead

# Structure – why?

(1) There is a lot of content to convey

(2) Humans have very small short-term memories

(1) + (2) => Provide detailed explanations of all your points. Describe all new terms comprehensively, going into all relevant details. Go back to previous concepts if a refresher is necessary.

# Structure – why?

(1) There is a lot of content to convey

(2) Humans have very small short-term memories

(1) + (2) => Provide detailed explanations of all your points. Describe all new terms comprehensively, going into all relevant details. Go back to previous concepts if a refresher is necessary.

(3) There is a page limit

(1) + (2) + (3) => You can't afford to waste any space, and you don't want to sacrifice clarity either. You need the reader to be able to follow your text and remember as they go. This can only happen if you divide your content into sensible and appropriately sized chunks.

# Structure (1)

- There's no fixed structure for this report (or for many others)
  - This gives you some freedom, but you still need to respect general structure guidelines, e.g. don't skip the introduction or the conclusion
    - In this coursework, keep introductions and conclusions short
- Use a separate section for each core part of your content
  - The stopping points should give your reader a chance to take a breath and summarise to themselves what they've just read
    - If you end a section too early, they won't be able to draw an appropriate conclusion
    - If you end a section too late, they may not realise you've moved on to a *different* subject

# Structure (2)

- Clearly delimit your sections, usually through a combination of spacing and bigger font for the heading
  - Keep the headings short, but use them to tell the reader what to expect to find in the upcoming section
  - For short reports, it's better to get straight to the point and avoid signposting
- 3–5 sections is a sensible default choice
  - At two pages, use subsections only if you're confident they add value
- Use plenty of paragraphs to allow your reader frequent breaks
  - If in doubt, (in academic writing) using more paragraphs is better than using fewer

# Layout (1)

- At 10–12 pt, a single-column layout may make your text hard to read
  - As a rule of thumb, don't exceed 20 words per line (but even 15 may be uncomfortably long)
  - This is why many academic papers are written in two columns
- If you decide to use two columns, be careful laying them out
  - Readers expect to go through the whole left column first, and only then move to the right column
  - Section breaks should be contained *within* a column

# Layout (2)

- You have many choices for embedding figures and tables in your text
  - Pick the right layout based on their sizes
  - In two-column layouts, they can span both columns, but make sure you don't alter the column flow
  - Don't use more white space than necessary around your figures
- **Always** caption figures, tables, code snippets, and all other floating elements
  - Number each element, then reference it in the text
  - Figure captions go *below* the image; table captions go *above* the table



# Layout: Examples

## Introduction to High Performance Computing Assignment 1 - Serial Optimisations

2023-24, 2024-25, 2025-26

### Introduction

This report chronologically follows the changes I made to optimise the `stencil` function. The time/speedup presented after implementing each change is the median average from three tests, to limit the effect of outliers. For the sake of time efficiency, initial tests will only be performed on the 1024x1024 image. Tests on the larger images, 4096x4096 and 8192x8192, will be performed once the baseline time of 1024x1024 has been reached. The following baseline is from compiling the unaltered `stencil` file with the unaltered Makefile and default GCC compiler, gcc-4.8.4, on Blue Crystal Phase-3, from here on referred to as HPC3.

compiler	image size	time /s
gcc-4.8.4	1024x1024	36.291698

### Initial compiler choice and flags

My optimisation strategy starts and finishes with changing the compiler and compiler flags, since they take little effort to change and may implement later manual changes. The table below details a selection of the compiler and flag combinations tried on the 1024x1024 image.

compiler	flags	time /s	speedup /s
gcc-4.8.4	-O0	36.2916977	1.000
gcc-4.8.4	-O1	16.8177990	1.131
gcc-4.8.4	-O2	16.7788311	1.131
gcc-4.8.4	-O3	16.8111195	1.131
gcc-4.8.4	-O3fast	32.2467692	0.900
gcc-7.1.0	-O0	36.2418190	1.000
gcc-7.1.0	-O3fast	32.3671660	0.931
icc-11.0.1.2	-O0	35.6112393	1.027
icc-11.0.1.2	-O3fast	32.1283492	0.986

Evidently, the default optimisation level for gcc is `O0` but more interestingly the flags `-O1` - `O3` all deliver similar times, something must be preventing aggressive optimisation. The `-O3fast` flag however seems much larger, speedup leading to the conclusion that by disregarding strict standards compliance, which may result in inaccurate floating point calculations, the compiler can optimise heavily. Since the python check script still passes, the `-O3fast` flag can be used. With this flag the intel compiler performs significantly better than GCC, perhaps unsurprisingly as HPC3 runs on Intel Xeon E5-2670. Unsurprisingly, because this allows the trade secret design inside the CPU and can write their compilers to optimise for it.

These tests are by no means exhaustive, since there are an enormous amount of compiler and flag combinations. Perhaps a solution to this could be a cloud service that

users can upload a file to, which then compiles with multiple compilers and flags, measuring the execution time for each compilation, and then reporting the best combination. Unfortunately such tools are not at my disposal and Intel compiler flags such as `-xAVX` which target AVX SIMD instructions will not have any effect until changes are made to enable vectorisation. Therefore compilers and flags will be revisited once all other manual changes have been made.

### Access pattern

In the inner loop of the `stencil` function, the index of the cell to be calculated is given by `j+i*imrow`. This means each iteration of the inner loop jumps to an index my access from the previous. Therefore the maximum number of operations is not performed in each cache line, as the same cache line will need to be revisited multiple times. Replacing the index access in the inner loop to `pointer+i` on the insight swapping the inner and outer for loops, makes the accesses contiguous. This leads to a time of 16.269909, i.e. a 7.28x speedup and achievement of the baseline time. This makes some considerations of its memory bandwidth, and as this change dramatically reduces the number of cache lines to be loaded.

### Vectorisation

An inspection into the vectorisation report produced with the `-qopt-report=5` flag reveals the main loop has already been vectorised. This is because the compiler has already realised the loop can be vectorised and enabled it. The report also states that the conditionals have been 'hoisted' from the loop. While removing conditionals from loops is required to enable vectorisation, it is also general good practice, as conditionals can cause branch mispredictions which lead to costly flushes of the pipeline. Further, adding the `-restrict` keyword to the image pointer arguments assists to the compiler that no aliasing is taking place, i.e. only one pointer points to each image, removing the possibility of vector dependencies. Making these changes manually leads to no change in time, confirming that the compiler is doing this itself. Closer inspection of the report reveals another issue, one of memory alignment, which will be tackled after changing from single precision to double precision.

### Data types

Since the main work to be calculated to perform the stencil of the 4-neighbours can be replaced with floats, i.e. the extra precision is not needed, then a speedup can be expected. Floats are 32-bit vs 64-bit and floats are 4-bytes meaning the same information will be stored in half the memory, meaning fewer costly loads are required. It is also important to multiply

Although a major component of producing a fast executable is the selection of an appropriate compiler and choice of flags, I've decided to leave the comparison of compilers until after I've produced my optimised code. My reasoning is that each compiler will perform differently on differently structured code, so comparing the performance of each on the unmodified code will not necessarily provide information that is relevant to my optimised code. It is entirely possible that the compiler which performs best on the unmodified code is actually less efficient than the alternative when compiling my optimised code.

The first and simplest modification I made to the supplied code was to change the datatype used from a double precision float to single precision. My reasoning for this change was that since the output image is rounded to some integer value in the range 0 - 255, there was no need for 64-bits of precision when compiling values. By changing the precision used, we can improve both compile requirements since adding and multiplying a 32-bit float is more compute intensive than a 32-bit float, and reduce the strain placed on the available memory/bandwidth since the variables being moved around and stored take up less space, and more of them can fit on a single cache line.

Next, wherever possible in the code I've altered it to operate on entire rows of continuous data rather than jumping around between entries in the array. One example is where the supplied code increments each pixel by the values of each of its neighbours in turn, before moving to the next pixel. In contrast, I've chosen to increment every value in a row of pixels by its vertical neighbour, since this allows the compiler to modify entire vectors of data with a single step, rather than using `if` operations to add each cells neighbours to it. Likewise, I add each pixels left, neighbour to it before moving on to right hand neighbour etc. .

Rather than using `if` statements to check if each pixel falls on the edge of the image and thus does not have a given neighbour, I've structured my code such that this is never an issue. This led to a much more complex code structure (and one that I certainly wouldn't appreciate if I was tasked with deciphering it) but removes the need for computationally expensive `if` statements, which form branch points and hinder the compilers ability to perform optimisation.

Another obstacle to efficient vectorisation that I've removed is the task of multiplying each neighbouring pixel value by 0.1 before incrementing the pixel. I found that my code ran much faster when neighbours were simply added unchanged (with the original pixel value scaled up) and the final product divided by a factor of 10 at the very end to keep values in the intended range. This allows the compiler to better vectorise the code since it is only adding the values of two arrays together and storing in one location, rather than multiplying one value by a scalar then adding that product to another array value, which cannot be done in a single step.

Gradually, I've added the `-restrict` flag to the pointers for both `temp_image` and `image`, telling the compiler that neither of them ever alias to any other variable pointed to by the other. This allows the compiler to safely vectorise without worrying about data dependencies between them that could cause incorrect results. Without this flag, the compiler would not have been able to vectorise almost any of the operations performed by the program since these two arrays are so central to the program's operation.

Finally, I've modified the code to account for the fact that the vast majority of the image goes completely unchanged in any given iteration of the `stencil` function. In practice, on the 10th iteration of this function, only a band of `n` pixels around the border of each square in the checkerboard will ever have a neighbour of a different value. This means that on the first iteration of the 8000x8000 setup, over 90% of the pixels are surrounded exclusively by pixels with the exact same value and thus do not need to be updated. By changing the code to only update this gradually widening band around each square, we can greatly improve the performance of the program at scale. In all honesty, modifying the code in this way did not provide as substantial a performance benefit as I'd hoped, I would attribute this to the fact that it massively increased the complexity of the program - lessening my ability to effectively identify inefficiencies amongst the somewhat tangled layout. I view this as a great example of the dangers of trying too hard to optimise

# Content: Story

- Write your report such that text *flows* naturally from one section to the next
  - Some call it *telling a story*, because connections between sections/paragraphs should be self-evident
    - But don't include irrelevant details—these just waste space and don't earn you any marks  
*I reverted my changes back to the last correct code commit*
  - Often this means presenting your content in non-chronological order
- Many CS research papers are similar in style, so reading some is a good point to start
  - <https://arxiv.org/archive/cs>
  - But do use your own filter: a lot of papers also have mistakes and bad writing

# Content: Style

- Remember this is *academic* writing
  - Don't use informal language  
**I thought it would be worthwhile** to run a vectorisation report with gcc to **get an idea** of what was limiting the vectorisation.
  - Avoid pompous words and prefer clear, concise, simple sentences
  - Use the appropriate technical terms and be aware of subtle differences:  
*GCC* is the name of a *compiler suite*, gcc is a terminal command
- It is common to want to chain sentences using commas
  - Don't. Stop as often as possible. Delimit statements clearly. Link them up with connectives that clearly point out coherence relations. Remember that your readers only have very limited scratch-space memory.
  - It is better to use a semicolon than to comma-splice, but abuse it and the pacing will upset your readers

# Content: Present Findings

- The purpose of the report is to present what you've done
  - Without it, we won't know, even if you *have* done the work
  - Include every detail that you think you deserve marks for
  - Back up **all** claims with evidence  
The code ran twice as fast, I **imagine** due to a higher level of vectorisation
- The report *shouldn't* contain raw data
  - Make good use of tables and graphs to *interpret* the data in intuitive ways
  - No screenshots—extract the relevant content and explain it
- Don't repeat yourself
  - Don't show the same numbers in *both* a table and a graph

# Content: Experiments (1)

- Take each of your experiments and present them individually

## Example: compiler experiments

- Clearly describe your set-up:  
I'm comparing GCC versions \_\_ and \_\_ with Intel version \_\_
- Do not change more than one variable at a time, because you might introduce confounders:  
I'm comparing **GCC with -O2** against **Intel with -O0**
- Quantify performance gains using speed-up  
The version compiled with Intel is 3.5X faster than the GNU one
- Explain *why* you are seeing these results:  
The compiler optimisation reports show that the Intel Compiler is able to vectorise my loops, but GCC is not

# Content: Experiments (2)

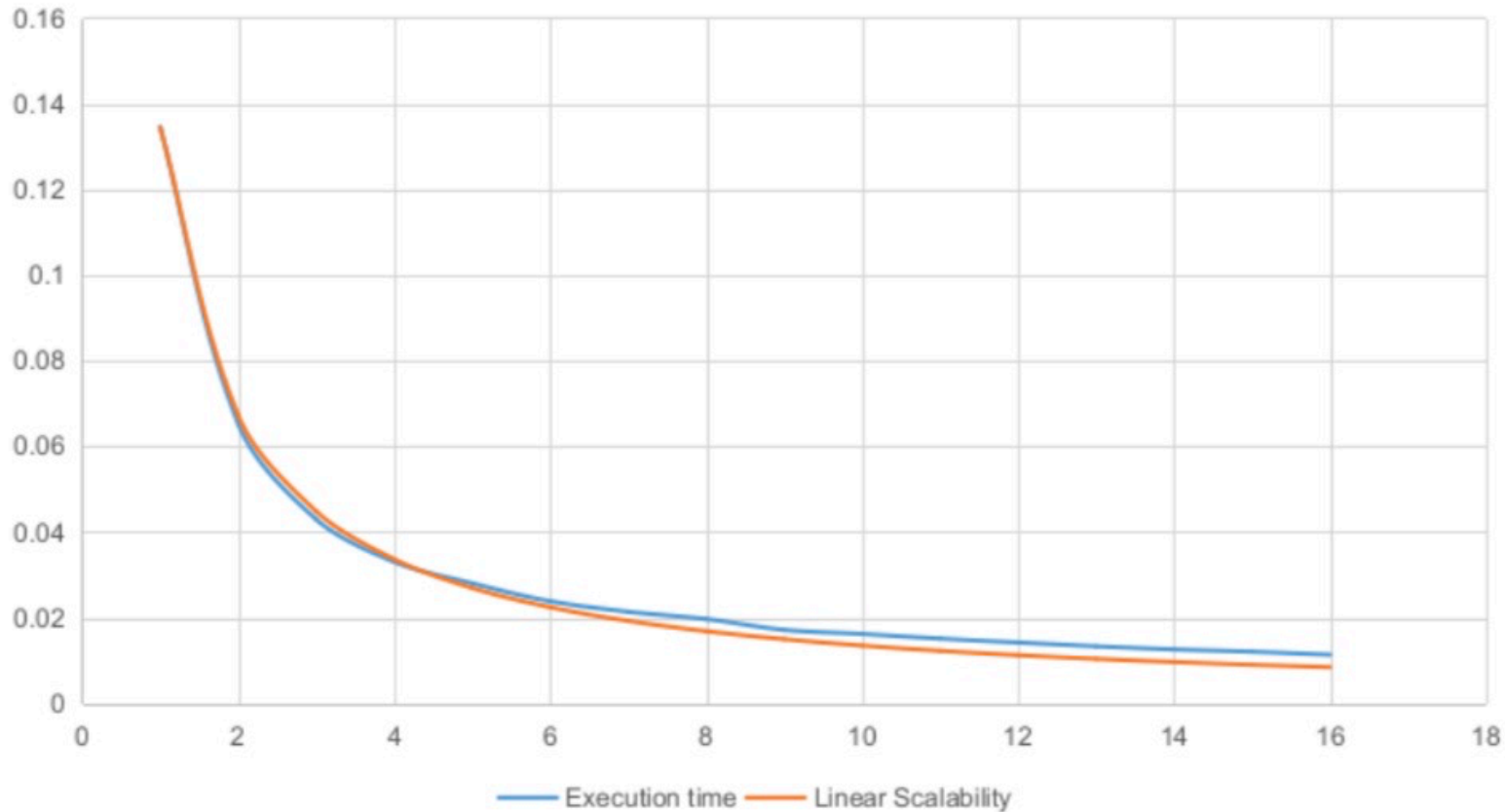
- If you have several data sets, e.g. different inputs, evaluate your work on *all* of them
  - For the stencil program, each input image will highlight different performance characteristics  
**All runtimes in this report will be based on an input of 1024x1024**
- Don't talk about work you don't have to do in the first place  
**I used a profiler to analyse the code and see where the biggest bottleneck was. The result was that the `stencil` method was taking over 99% of the time of the program.**
- When presenting results, choose the right units and precision:  
**The run time decreased to 00:00:00**  
**My final run time was 0.85 s**
- Be careful of noise:  
**My run time improved from 3.23546 to 3.23518 s**

# Content: Figures

- Pick the right type of graph for the data you are showing
- Always have a legend, always have labelled axes, always show the units you are working with, always caption figures
- For data points that are far apart, consider using a log scale
- Choose a sensible colour scheme, avoiding similar tones
  - Keep in mind some readers may be colour-blind
  - If you need to project your graphs, light colours are risky



# Figures: Examples (1)



This graph has many issues:

- The data series is discrete, i.e. no real data was recorded for non-integer number of cores, so the data points need markers
- No axis labels
- No units
- $x=1$  is an important data point and it's not clearly presented
- x axis doesn't need to go up to 18

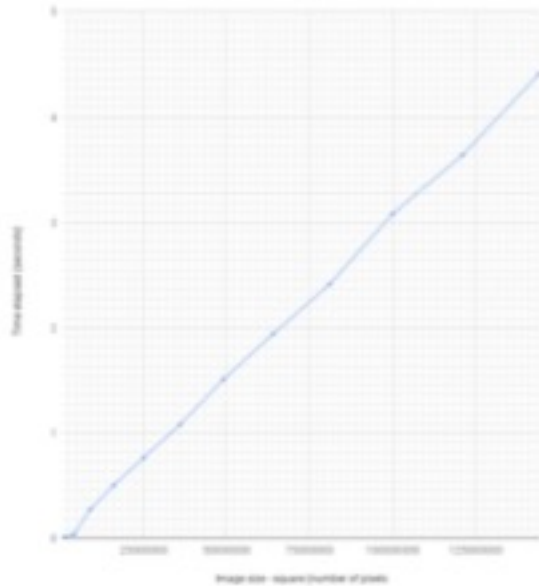
# Figures: Examples (2)

ppn=1	ppn=2	ppn=3	ppn=4	ppn=8	ppn=16
runtime: 0.105461 s	runtime: 0.054993 s	runtime: 0.036725 s	runtime: 0.029283 s	runtime: 0.016261 s	runtime: 0.010369 s
	runtime: 0.055470 s	runtime: 0.037238 s	runtime: 0.029787 s	runtime: 0.016328 s	runtime: 0.009788 s
		runtime: 0.036957 s	runtime: 0.029389 s	runtime: 0.017388 s	runtime: 0.023223 s
			runtime: 0.029011 s	runtime: 0.016657 s	runtime: 0.010701 s
				runtime: 0.017182 s	runtime: 0.009990 s
				runtime: 0.016947 s	runtime: 0.009883 s
				runtime: 0.018317 s	runtime: 0.010898 s
				runtime: 0.016482 s	runtime: 0.009698 s
					runtime: 0.009672 s
					runtime: 0.010788 s
					runtime: 0.010604 s
					runtime: 0.010463 s
					runtime: 0.010285 s
					runtime: 0.010199 s
					runtime: 0.010106 s
					runtime: 0.009604 s

Figure 2: Run time for 1024x1024 when using 1, 2, 3, 4, 8 and 16 cores

- This is not a figure—it's a table
- In the original report, this is a *screenshot of a table*
- More than half the page space taken by this object is wasted
- There are too many decimals shown: the runtime is 0.01 s + noise
- Tables should be captioned at the top, not at the bottom

# Figures: Examples (3)



(a)



(b)

- This graph is taken full-resolution from a paper
- You can't read the text even on a monitor (where you can zoom), let alone on paper
- Make sure the text on your graphs is readable—print out a copy if you're unsure

# Content: Explain Findings

- It is not enough to show *what* you've done
  - Remember that you are trying to demonstrate understanding
  - You need to explain *why* the effect your are observing occurs
    - This applies for both positive and negative results
- Be careful with concepts you don't have a clear grasp of...

The conditional **if** statements were replaced by **for** loops, which are considerably **less expensive** and **avoid branching predictions**.
- ... and with misusing technical terms ...

**Restricting pointers** might also help with vectorisation.  
Conditionals in a for loop **break the pipeline** in the processor.
- ... they both work against you

# Content: Language

- If your writing is riddled with language mistakes, it is natural for the reader to suspect the rest of your work may also be flawed
  - In writing—as in many other aspects of life—first impression *does* count
  - A proofreading pass or two go a long way to improving the perceived quality of your write-up
- Common mistakes (from past years) to be aware of:
  - *its* != *it's*
    - even though it may feel natural to apply the same transformation as in *Joe* → *Joe's*
  - Semicolons (;) **never** go before lists
    - that's what colons (:) are for
  - Commas **never** separate a sentence's subject from its predicate
    - even if it's natural to pause when speaking

# Proofreading

- Read your report **many times**
  - Ask yourself whether *your reader* will understand (as opposed to whether *you* understand)
- Read your report out loud
  - If you find it hard to *say* the words, they will not be pleasant to read either
  - If you run out of breath, you need to split up your sentences more
- Don't rely on your editor's spell-checker only
  - You can try reading your sentences *backwards* when you check for typos

# Conclusions

- Leave enough time for your report
  - This is what you are marked on!
- From now on, you'll only need to do *more* writing, so use this opportunity to practise
  - We'll give you individual feedback on your submissions
- Think about and avoid the common mistakes
- Write clearly, concisely, unambiguously



Questions