Thinking in Parallel

Finding parallelism, some parallel patterns, implementing these on Viking

1. Motivation
2. Finding parallelism
3. Pattern 1: SPMD
4. Pattern 2: Loop Parallelism
5. Conclusions

Motivation - what is it?

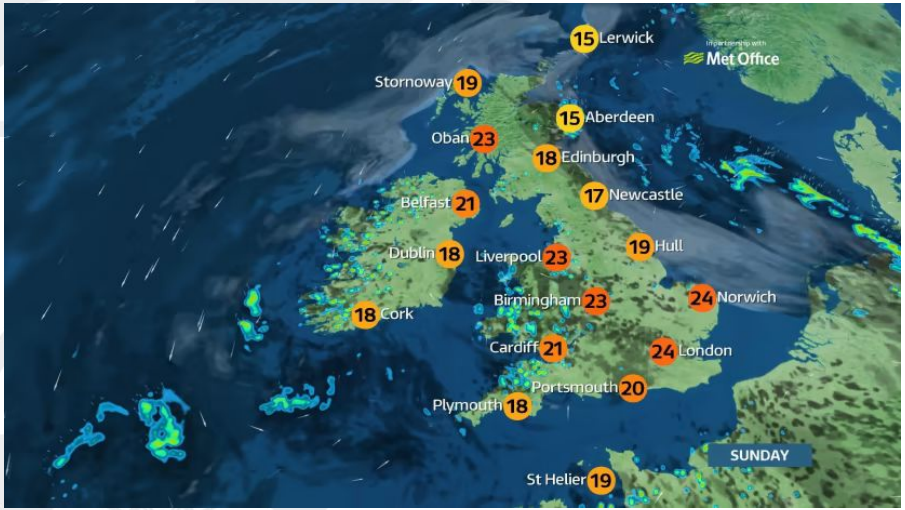
- Units of work are completed **at the same time** as each other
- Requires hardware support
- Contrast with *sequentialism* - units of work are completed **one at a time**, one after the other
- We can find and exploit potential parallelism where we typically think sequentially

Motivation - why bother?

- Parallelism is baked in to the hardware on which we are carrying out computation!
- From consumer-grade hardware to national supercomputer class systems, parallel architectures are the standard
- If you're walking around with one of these pens in your pocket, you may as well make use of the black and red ink

Image credit: https://www.gopromotional.co.uk/

Motivation - why bother?



- Solving problems *at scale* - much more than a phrase du jour of enterprise IT!
- As problem sizes increase, so too must the elegance and efficiency of our solutions
- Some questions may only practically be answered with parallel strategies
- Especially applicable to operational problems - the weather forecast comes to mind

Image credit: https://itv.com/

Motivation - why bother?

- Why do in 10 hours what can be done in 10 minutes?*
- Nobody likes to wait for work to complete - it breaks the reward loop, and dulls motivation / excitement
- There are often low hanging parallelisation fruits (I'm looking at you, `for` loops) that can satisfy our lust for speed

* I make no guarantee of this kind of speedup

Image credit: https://fandom.com/

Motivation - take care!

- Just because you can, doesn't mean you should
- Computation == energy, energy usage has tradeoffs
- Hardware manufacturers progressing towards energy efficiency
- Programmers are more of a mixed bag!
- Computation at scale can have a significant impact*
- Start small, validate, run minimally

* See https://arxiv.org/abs/1906.02243 as an example

Finding parallelism - how hard should we try?

- Gene Amdahl, 1922 - 2015
- Designed the WISC, an early digital computer (6K of memory and 60 operations a second in the early 50s!), for his PhD thesis…
- Responsible for significant architectural developments at IBM (System/360 very successful mainframe system)
- Eventually formulated “Amdahl’s Law”

Image credit: https://www.i-programmer.info/history/people/300-gene-amdahl.html

Finding parallelism - how hard should we try?

UNIVERSITY of York

Amdahl's Law:

$$S \leq \frac{1}{(1 - f) + \left(\frac{f}{p}\right)}$$

- Speedup (*S*) is bound by the fraction (*f*) of the program which is parallelizable and the degree (*p*) to which it can be parallelized
- As *f* tends to 1, speedup is bound only by *p*, and *p* is bound by practical limitations!
- At large *p*, speedup is dominated by (1 - *f*), the fraction of the program which *cannot* be parallelized
- We can't necessarily throw resources at the problem to make it Go Fast, and might need to rethink
- Note the LTE symbol - there are considerations beyond this which will inform potential speedup…

Finding parallelism - how hard should we try?

UNIVERSITY of York

- Important to spend time understanding the problem:
  - Guards against potential wastefulness
  - Points you towards reformulating the problem if you discover immovable serial sections
  - Sometimes things do have to run in serial, so set it running and write some tests & documentation…

Finding parallelism - where to start?

UNIVERSITY of York

- Two broad concepts of parallelism to consider:
  - Task-first parallelism - identify program sections that could be split into independent ‘tasks’, which can operate at the same time
  - Data-first parallelism - identify subsets of data which can be independently operated on by your program to solve a larger problem
- For many, thinking about data-first parallelism can provide the quick wins we crave
- These two approaches overlap - tasks need data, data needs tasks to operate on it!

Finding parallelism - where to start?

UNIVERSITY of York

In either case, we should consider the following:

- **Flexibility**: allows the design to be adapted as requirements change - not sensible to dig yourself into a trench early
- **Efficiency**: parallel programs are useful if they make efficient use of the resources provided to them
- **Simplicity**: a parallel solution ought to be sufficiently complex to achieve its goal, but needs to be debugged and maintained!

Finding parallelism - task-first considerations

- Need to neatly encapsulate tasks into callable units
- Need to balance work done by parallel tasks with the amount of work needed to manage their existence!
- Need to evenly distribute work amongst parallel tasks else suffer poor parallel efficiency*
- Consider the unit of work represented in a task: individual function, iterative construct etc. - these can be implemented differently (more later)

*** There's a whole world of work on this! To begin with, start with a homogeneous work distribution and cross the jagged bridge if/when you get there…**

Finding parallelism - task example

- Imagine a study of axe throwing skill with increasing alcohol consumption, many participants (no Amazon voucher needed)
- Calculating correlations and fitting linear regressions across many participants in a large file
- Correlations and regressions can be calculated independently!
- Can synchronise at the end to compare correlation and regression coefficients, to produce a summary
- Being able to efficiently process means we may have more time to iterate on analysis techniques

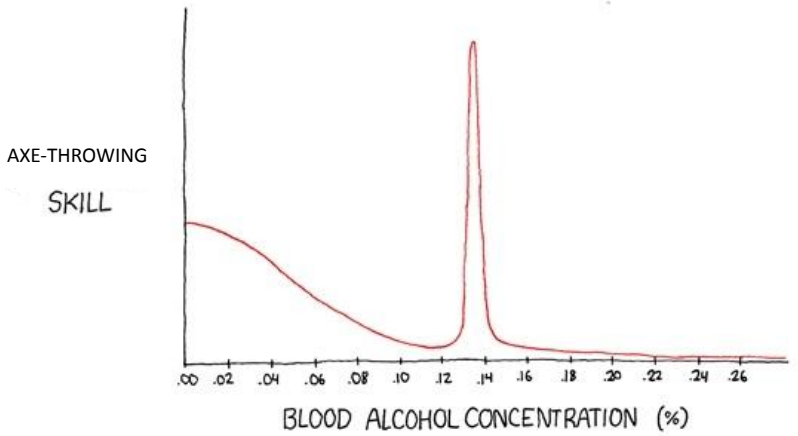


Image credit: https://xkcd.com/

Finding parallelism - data-first considerations

- Is the problem organised around manipulating a large dataset?
- Are we applying identical / similar operations to subsets of a dataset?
- Can the dataset manipulations be carried out independently?
- Do you want to be able to run this on a range of systems (e.g. sometimes on laptop, sometimes on Viking)?

Finding parallelism - data-first example

- Imagine applying smoothing to a massive, high resolution image
- Achieved using a 2D Gaussian filter - first convolve image rows, then convolve image columns
- We could sequentially iterate through the convolution - this might take a while…
- We know that we can operate on slices of the image independently, and will need to synchronise a couple of times
- Divide image up into appropriately-sized slices, then consider task parallelism

Image credit: https://www.mathworks.com/

Finding parallelism - constraints

- We need to think about which of our tasks can be grouped together:
  - To identify constraints - do tasks need to synchronise? Do some tasks need to share data? Can groups of tasks run concurrently to improve parallel efficiency?
  - A logical task grouping simplifies your experience with the program
- We need to think about the order in which our tasks must execute:
  - Does *X* need to happen before *Y*? Do some tasks require online information from others, requiring simultaneous execution? Is there any ordering at all?!
  - This is often straightforward to intuit from a solid understanding of the high level problem
- We need to think about which tasks can / must share data:
  - Especially important when working on large problems - I/O is expensive
  - Data access optimisations
  - Coupled with grouping and ordering - can data only used by some group when another group is finished?
  - Do we *really* need to share data? Communication between tasks is also expensive!

Finding parallelism - conclusions

UNIVERSITY of York

- Am I confident that I understand the problem?
- How much can actually be parallelised?
- Am I working with a large dataset over which I do lots of the same things?
- How can I group, order, and share data between tasks?
- How straightforward is it to work with my parallel program?

Pattern 1: SPMD

- “Single Program, Multiple Data”
- We have a program (our task) that operates on some data - initial condition, dataset, whatever - and we’re **happy with it**
- We’ve identified that many copies of this task can run **independently of each other**
- Where applicable, we’ve identified anywhere tasks need to synchronise due to ordering constraints

SPMD - how can we achieve it?

- If we don't need to synchronise, we can make use of our system's workload manager to handle the setup of many parallel tasks
- If we do need to synchronise, we should consider an established supporting tool, e.g. MPI
- This is a common parallel pattern - you will find it everywhere!

SPMD - pros and cons

Pros:

- Scales well up to previously mentioned workload distribution limits
- A simple mental model - lots of the same program, with a bit of go between
- Overheads of parallel task management are relegated to the beginning and end of the program

Cons:

- As program content becomes more complex, simplicity of mental model degrades
- Managing program outputs can be messy, and I/O at the scale this approach allows you to reach becomes a dominating efficiency problem
- Need to think carefully about the amount of communication between tasks

SPMD - workload manager example

```
#!/usr/bin/env bash

#SBATCH --job-name=ARRAY_EXAMPLE
#SBATCH --account=MY-PROJECT-2022
#SBATCH --partition=nodes
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=50M
#SBATCH --time=00:01:00
#SBATCH --output=%x_%A_%a_EXAMPLE.log
#SBATCH --array=1-4

echo "Hello from ${SLURM_ARRAY_TASK_ID}!"
```

- We want to run our task, *echo*, in an SPMD configuration - many copies of the program but with different initial data (the task ID)
- We specify the resources needed to run a single task (*--ntasks=1*)
- We ask output to be written to a file that is a combination of the job name (*%x*), the overarching job ID (*%A*), and the task id (*%a*), separating the output by task
- We ask Slurm to set up 4 copies of the task to be run in parallel (*--array=1-4*)
- We can access our task ID with an environment variable (*SLURM_ARRAY_TASK_ID)*

SPMD - MPI example

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
          MPI_Init(NULL, NULL);

          int world_size;
          MPI_Comm_size(MPI_COMM_WORLD, &world_size);

          int world_rank;
          MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

          printf("Hello world from task %d!\n", world_rank);

          world_rank = world_rank + 1;

          int* world_ranks = malloc(sizeof(int) * world_size);
          MPI_Allgather(&world_rank, 1, MPI_INT, world_ranks, 1, MPI_INT, MPI_COMM_WORLD);

          int rank_sum = 0;
          for (int i = 0; i < world_size; ++i) {
                    rank_sum = rank_sum + world_ranks[i];
          }

          printf("Task %d has computed a sum of %d!\n", world_rank, rank_sum);

          free(world_ranks);
          MPI_Finalize();
}
```

- We want our tasks to do something independently (say “Hello…”), then communicate (*MPI_Allgather*) a result (*world_rank + 1*) with each other for further processing (calculating the sum)
- We initialise a “communicator” - collection of tasks which can talk with each other
- We can carry out independent work as we would outside of the parallel context
- MPI provides mechanisms for tasks to talk with each other
- We can access our task ID via the “rank” property - one way to allow tasks to behave differently
- Careful of `if (rank == ...)` spaghetti!

SPMD - MPI example

- In this case, we want to run several copies (*--ntasks=4*) of our task (*MY_PROGRAM*) with MPI
- We need to make sure that we've loaded the right modules
- This time, the MPI runtime will be managing our SPMD program instead of Slurm
- We can either run this through Slurm (*srun … --mpi=)* to benefit from more granular reporting OR run through *mpiexec* to benefit from portability
- In this case, all output goes to the same file! Have fun sorting through it for a large number of tasks...

```
#!/usr/bin/env bash

#SBATCH --job-name=MPI_EXAMPLE
#SBATCH --account=MY-PROJECT-2022
#SBATCH --partition=nodes
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=200M
#SBATCH --time=00:01:00
#SBATCH --output=%x_%j_EXAMPLE.log

module purge
module load toolchain/foss/2021b

srun -n "${SLURM_NTASKS}" --mpi="pmi2" ./MY_PROGRAM

# OR

mpiexec -n "${SLURM_NTASKS}" ./MY_PROGRAM
```

Pattern 2: Loop Parallelism

- Exactly what it says on the tin - exploiting potential parallelism in loops
- We have a serial program whose structure is dominated by computationally intensive loops
- We think that the loop iterations could work mostly independently of each other
- We think loop iterations are intensive enough to justify the overhead of managing parallelism

Loop Parallelism - how can we achieve it?

UNIVERSITY of York

- If any dependencies exist between loop iterations, rework loops to minimise these
- Employ one of the **many** loop parallelism support libraries that exist to do it for us:
  - C, C++, Fortran - OpenMP
  - Python - multiprocessing, Joblib
  - R - foreach
  - MATLAB - parfor

Loop Parallelism - pros and cons

Pros:

- Usually a simple bolt-on that will get you parallelism with little investment
- Can often modify existing programs without concern for semantic changes
- Availability and simplicity of support libraries

Cons:

- Scaling limitations - need to employ additional mechanisms to scale beyond your machine
- Feels the full force of Amdahl - if you have few or non-intensive loops, you are unlikely to see significant performance gains
- Can stop you from seeing the forest for the trees

Loop Parallelism - Python example

```
import glob
import os
import typing

import joblib
import matplotlib.pyplot as pyplot
import pandas

def plot_values(
  data: typing.Tuple[str, pandas.DataFrame], columns: typing.Tuple[str, str]
) -> None:
  figure, axes = pyplot.subplots()
  data[1].plot(x=columns[0], y=columns[1], ax=axes)
  plot_name = os.path.splitext(os.path.basename(data[0]))[0]
  figure.savefig(f"PLOTS/{plot_name}.png")

if __name__ == "__main__":
  FILENAMES = glob.glob(os.path.join(os.path.realpath("./DATA"), "*.csv"))
  DATAFRAMES = [
    (FILENAME, pandas.read_csv(FILENAME)) for FILENAME in FILENAMES
  ]

  # PLOT IN A REGULAR LOOP:
  for DATAFRAME in DATAFRAMES:
    plot_values(DATAFRAME, ("field_1", "field_2"))

  # PLOT IN A PARALLELISED LOOP:
  joblib.Parallel(n_jobs=-1)(
    joblib.delayed(plot_values)(DATAFRAME, ("field_1", "field_2"))
    for DATAFRAME in DATAFRAMES
  )
```

- We have a bunch of potentially large data files (**.csv*) from which we would like to plot two data fields against each other
- We could loop through them and plot them one at a time, this can be slow (especially with Matplotlib!)
- We import the joblib module (*import joblib*)
- We know that plotting tasks can occur independently of each other
- We encapsulate the plotting in a function (*def plot_values*) for convenient calling
- We use the *joblib.Parallel* construct to execute our loop over as many cores as we have access to (*n_jobs=-1*)
- Still have a potentially substantial serial section, depending on how tricky it is to read in data

Loop Parallelism - Python example

- We want to run our loop-parallelised program (*my_script.py*) so need to request more than 1 core
- We are doing one thing, so *ntasks* is now 1
- We want that one thing to have access to 4 cores (*--cpus-per-task=4*)
- Assuming you have a Python environment set up with the modules you need, can just run Python!

```
#!/usr/bin/env bash

#SBATCH --job-name=LP_PYTHON_EXAMPLE
#SBATCH --account=MY-PROJECT-2022
#SBATCH --partition=nodes
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem-per-cpu=200M
#SBATCH --time=00:01:00
#SBATCH --output=%x_%j_EXAMPLE.log


python my_script.py
```

Conclusions

- Parallelism can be a great way to improve program performance and to scale to new problem sizes
- We should be careful about how much time and how many resources we throw at parallelisation
- There are many support libraries available to facilitate parallelisation
- Coding Club drop-ins are a great place to talk out potential parallelisation!

Resources

- *Patterns for Parallel Programming*; Mattson, Sanders, Massingill
- ARCHER training courses
- Coding Club Slack channel and drop-ins
- Experimentation and chatting with Viking support team