Parallelism Thought Experiments

Part 1 is a walkthrough of some example problems and which parallelism approaches suit them. Part 2 outlines some research problems and asks you to work this out. The aim of this is to think about how the various parallelism models we've discussed might be applied to a range of problems, to help you understand the benefits and challenges of each. Ask for feedback if you want - there are no "canned answers" we can provide

Part 3 covers some tips and questions on actually running parallel codes - how to select the resources you'll need, what is oversubscription etc

- 01 Comparing some problems

- We've given a few examples in the Examples directory of the same problem programmed in serial, MPI and OpenMP approaches
- Let's walk through a few of these and consider the benefits and problems of all the approaches

First example - The Collatz conjecture:

A conjecture in maths that, if you make the following sequence of steps, every starting number will eventually get you to the value 1. The steps are simple: if the number is even, divide it by 2. If it is odd, multiply by 3 and add 1

For example, 6 (÷2) -> 3 (*3 +1) -> 10 (÷2) -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

Suppose we want to check by exhaustion starting numbers from 1 to a 1000. As a parallel problem, this is easy! It's embarrassingly parallel - there is no connection between items to check (actually there is, see below). All we have to do is divide the numbers to check into blocks and assign one block to each processor.

So we can use GNU parallel, suitably set up, threads, OpenMP parallel loops, or MPI parallelism by dividing the numbers to check.

However, we've already actually hit 2 reasons why one method is better than others. **Firstly**, the sequences are highly variable in length (see https://oeis.org/A006577 for the number of steps needed) and so simply cutting the inputs into large blocks might not be very efficient in terms of using resources. Some processors will have

done their work and be sitting idle while others are still working. Fixing this is called "load balancing", and involves splitting the work into smaller blocks and trying to combine them together so they add up to the same total work.

Splitting into smaller blocks and then handing them out as needed might be enough, although we might find it challenging to do a good job. OpenMP becomes an obvious choice, as it can take care of this scheduling for us with no effort on our part, and has some abilities to hand out the work to try and balance the load

Secondly, there is actually a connection between subsequent values - if we ever hit a number we have already checked, we can stop - we know the answer without repeating that part of the sequence. This means we might be able to reduce the work significantly by making some sensible algorithmic tweaks.

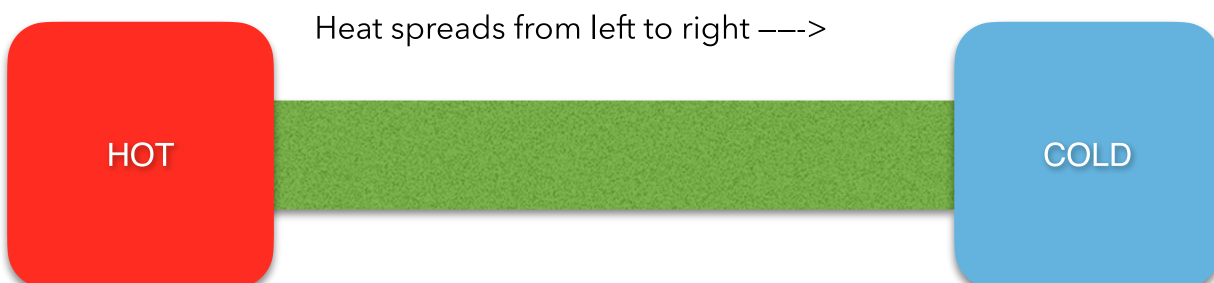
This is one of the most important things to remember when dealing with parallelism, or in fact programming in general - *no approach will ever be faster than simply not doing the work at all!* Always look for algorithmic shortcuts first!

Keeping a record of every value we have reached might be possible, although it might produce a lot of data. In this case we'd probably be best with a threaded approach, as this data is naturally shared. Checking every value we reach against this may or may not perform though.

On the other hand, we can get a bit of a bonus by saying "if we ever reach a value below the value we started with, another thread will have checked things from there and we can stop". This doesn't require us to keep any data around, so introduces no new shared information and is really *easy* to implement.

Second example - The heat equation:

Suppose we have a cold metal rod, and we put one end against something hot and the other against something cold (for the physicists, assume fixed temperature heat bath attached to the ends).

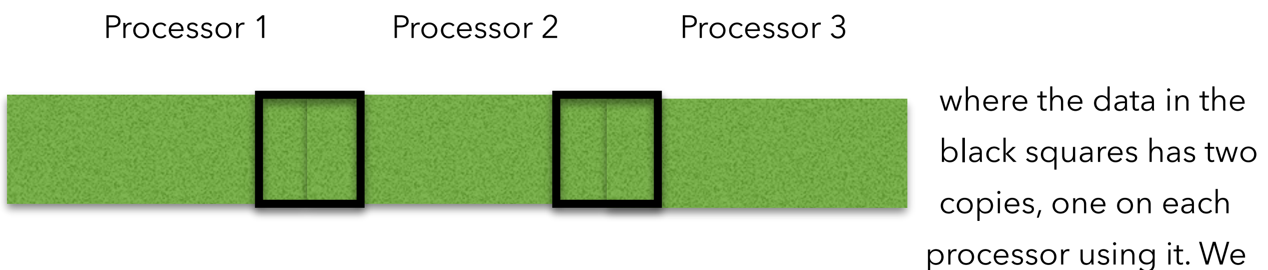


After some time, we reach a steady state where the temperature goes from hot to cold along the rod and we want to find the temperature at every point. We know that is going to depend on the temperature of the points to the left and to the right.

So our program is going to run some repeated calculation (iterating towards the final solution), where the work at each point needs information from the points around it. This is a fairly tightly coupled kind of parallelism, and GNU Parallel is quickly ruled out.

Threading or OpenMP will work - we have one shared array of the points at the current time, one for the points at the next time, and we split up the work of calculating the values. Since we use a temporary array for the updated values, we never have two things writing to the same data location, and its almost trivial to implement this.

MPI might seem a less obvious choice, as we'll have to split up the rod into pieces and share the values at the ends with adjacent pieces, like this:



where the data in the black squares has two copies, one on each processor using it. We have explicitly share those values at each step. This is the problem of "synchronisation"

However, in reality the 1-D heat equation like this is boring and we need to go to 2-D or 3-D to have an interesting problem. In 3-D things get interesting - if we divide up our space into 1024 points along each direction, we'll need about 10 GB just to store the data, and this is not really a very large system in engineering terms. At this point, MPI and distributed memory parallelism is really the only option. Buying more memory helps to a point, but a bigger simulation is always lurking just around the corner. We need to write code to exploit multiple nodes, and MPI is how.

By the way, in general terms computers have been growing such that the "local computer cluster" of ten years ago is todays "powerful workstation" and the "personal computer" of ten years ago is now a pocket calculator. Depending on your perspective, you can think of it as "with MPI I can do work today that without it I'd have to wait ten years for" or alternately "if I can just wait ten years I can do this without any clever programming"

- 02 Some more problems to consider

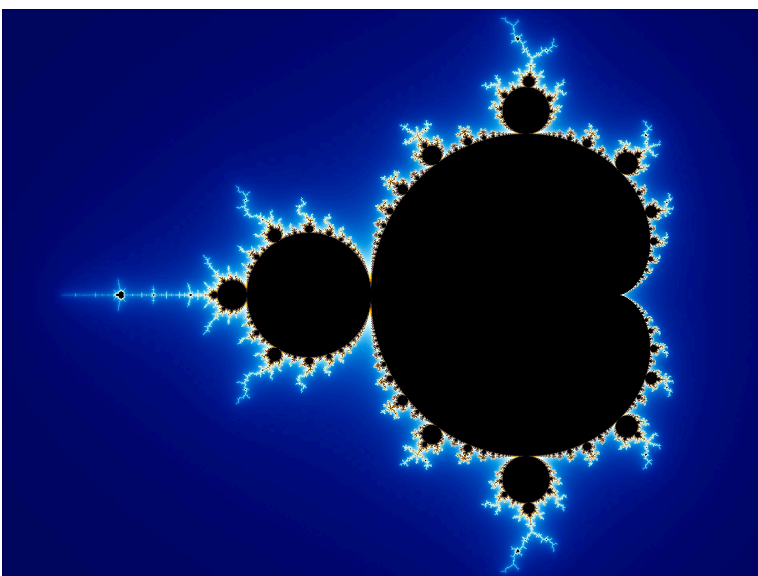
- Now we're going to describe some general problems, and its up to you to think about the challenges involved
- We're going to mostly use examples that are images, or that involve calculations with a lot of inputs to cover, since these are the easiest to describe
- Consider the challenges we mentioned such as synchronisation, race conditions, the extent of coupling (none, weak or tight) between items, and questions like memory requirements, and performance overheads

First example - recolouring a large image:

Suppose we have a rather large image file that we want to recolour according to some simple local rule. Perhaps our camera makes things overly red and we want to turn down the red channel. Thinking of an image as a series of independent pixels, how might we parallelise this action? Do you think it matters how complicated the calculation of the new colour is? The next example will explore this some more.

A real one - visualising the Mandelbrot set:

While recolouring an image pixel-by-pixel doesn't involve much calculation, there is a classic problem which involves colouring pixels according to a complicated calculation to visualise something called the Mandelbrot set.



(Created by Wolfgang Beyer with the program Ultra Fractal 3. - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=321973)

A point is part of the set if a repeating calculation never reaches a certain threshold, and it is not part of the set if the calculation repeats forever. Assuming we have some way of limiting "forever", this is a lot like the previous example, but where we have a difficult calculation on each pixel (see https://en.wikipedia.org/wiki/

Plotting_algorithms_for_the_Mandelbrot_set for details if you want to know). No pixel depends on the ones around it, so which parallelism approaches might we use?

Next example - Flight mapping:

Suppose we have a lot of flight data. We want to produce a map coloured according to the number of planes flying over each location. Since we're drawing a map, we'll want to set some granularity, say 1km. Each 1kmx1km square on earth becomes one pixel. Now we take our list of flight trajectories and, for each one, work out which pixels it will cross, and add one to the count in those (an accumulate operation). Which parallelism approaches would work here? How does your choice depend on the size of the region you're mapping and the amount of flight data? What about the complexity of the trajectory calculation?

A final one - doing a bunch of different things:

Extending the last example, suppose we also have a lot of satellite images of earth and we want to investigate the correlations between aeroplane fly-overs and cloud coverage and type. There's roughly 4 separate tasks here: the mapping as before, processing images for cloud coverage percentage, and for cloud type, and then correlating the two sets somehow. Which parts of this might be amenable to parallelism? What methods might you use? What if you had an extremely fast hard drive, such that reading the images from disk is easy compared to the computational tasks, would that change the viable approaches?

- 03 Running parallel codes

  - Once we have a parallel program, either that we have written or obtained, we want to run it.
  - How do we know what resources it can use?
  - How do we tune our resources to make sure we make best use of them?

First example - a parallel code we wrote entirely:

This is the hardest to write but the easiest to run in many senses. We know which bits we made parallel, and we know how the parallelism will respond to increased resources, so we should be able to make a good guess of how many threads or cores we can use for a given problem size. We should still verify that we're making good use of the resources we request, at least before we run multiple jobs or very large jobs. We might want to run some examples and plot ourselves a scaling graph to inform this.

Second example - a parallel package with good documentation:

Good docs are sadly rarer than they should be. But if they exist, they can probably answer all your questions about how much resource a code can make use of. Read them! In particular, look for information on how the number of threads or processors will depend on the size of job, and keep in mind that memory requirements may affect how big a job you can run. You'll also want to think about "time-to-solution" versus "cpu-time-to-solution". Perhaps you can run for longer, using checkpointing or by asking for a job time extension, rather than add resources.

Third example - a package we're not really sure about :

Sometimes the only way to tell if something can use resources is to try it. Pick a typical sort of job you want to run, and find a resource that works. Then try doubling the number of threads or cores and see if it takes half as long. Or, use a monitoring tool to see whether the processors are being used to 100%.

Fourth example - a mixed bag:

The trickiest situation is one where we have written some code that uses some packages behind the scenes which may be able to exploit multithreading, and we are considering adding threading or ensemble runs as well. We want to make sure that we do not "oversubscribe" resources. For example, if we multiply two matrices, a lot of languages will already be parallelising this operation - if we try and run 8 copies of this code we may find it only slows down the final operation. This is because the operating system will have to do all that task-switching to service the threads we've now asked for, and this costs us. Before adding your own parallelism, do check whether you could just give your libraries a flag and have them take care of it, and do try and understand which parts of your code this will affect and which might need more finesse.

TAKEWAY POINTS:

- If you can at least estimate the amount of cores, threads, memory etc you can refine from there by checking performance monitoring tools or trying a few sizes and observing the time taken
- Some problems are easy to parallelise, others are very hard. Sometimes it is still worthwhile, but only you can decide
- Making efficient use of resources on shared systems is polite, as these systems are rarely sitting idle. The more resource you need, the harder you should try to be efficient with it
- Sometimes parallelism can happen behind-the-scenes in libraries or packages, so before adding explicit parallelism, find out if you can get it "for free" already
- We want to help you make use of things so we all get the most out of the systems, but we don't know your research or the software you might be using. The better you understand parallel ideas, the more we can help you