Things to Try in Open MP

We have tried to keep these programs small, so we haven't written any functions etc! Please don't think this is a recommended way to write code!

Part 1 - Basic Programs

- 1A: Walkthrough of a first program

  - To get you started, this gives a walkthrough of writing a simple OMP program from a simple piece of serial code, FirstProgram.[f90|c]
  - This code sums the numbers from 1 to N in a loop. There is a known answer to this, which we print at the end to check we got the right results

Info:

Read over the sample code and make sure you understand it! We're going to add some basic parallelism, which will be a lot easier if the code is familiar. Compile and run the code too. Does it do what you thought?

Problem Description:

The first step is to add the USE or include line and make sure we can compile with -fopenmp. Tackle this before going on!

There is only one loop to parallelise here. Put in a OMP PARALLEL DO/FOR section. Don't worry about the answer yet! Just compile and run this.

The answer might seem to be right (it does on my machine). If it seems right for 100 items, try a larger number, say 10000. Can you guess the problem?

This is one major pitfall of OMP - sometimes an error only occurs for some values, so we always need to check a few before concluding we're done.

What we've done has a big problem, just like mentioned on the slides. The *sum* variable is shared and we can get collisions and a wrong answer from the accumulation line. This isn't terribly likely, and so might only occur every-so-often. We can increase the chance of seeing it by increasing T or doing a lot of runs.

Adjust your OMP do loop to do a REDUCTION on the sum variable using the '+' operator. Check the answer now. If it was wrong before, it should now be right!

- 1B: A Harder program

- Now you've done some OMP, its time to try it without full guidance but on a similar, simple piece of code
- We're going to use a simplified version of a classic problem - the grid automaton, or Conway's "Game of Life" (https://en.wikipedia.org/wiki/Conway's_Game_of_Life)
- This has a grid of cells, whose state is updated over time according to some simple rules. The interesting thing is that these rules can lead to propagating structures and ordering which is more complex than the rules - a sort of emergent complexity. We have set up a simple "glider" structure, which glides across the grid over a few time steps, always returning to the same shape
- Also, its a simple code with an interesting outcome. We've provided a serial version in C or Fortran. This solves a small grid (10x10) with output to the console. The file is called GameOfLife.[f90|c]

Info:

Again, read over the sample code and make sure you understand it! Compile and run the code too. Does it do what you thought?

When running larger grids, you will want to disable the output to screen as it will be useless and take a lot of time.

Problem Description:

There are 3 nested loops in this program. The time loop is NOT suitable for paralellising, because each iteration depends on the last.

Try adding the simplest paralellism - either the ix or the iy loop. Pay attention to which variables need to be private! Run this on a small grid and verify the answer is unchanged on 1, 4 and more (if possible) processors.

Now turn off the print to screen and try running on a much larger grid. On our 16 core machine, we had to go to around 10,000x10,000 to get good performance and scaling

Fortran only - Like the image blurring example, we have to copy the updated grid back. We're possibly losing a lot of performance by serialising this. Try parallelising it. Does it perform better now?

C only - We used a pointer swap to avoid copying the grid, so we already have close to the best case. As an experiment, replace this with a loop and observe the performance loss. Assuming we can't do tricks with pointers, try parallelising this part too. Does this help restore performance?

Extension options - There are several ways to extend this program to be much more useful, which also lead to trickier paralellism. Try any of the following:

1. Instead of the glider, assign each cell a random initial value. You'll need a way to generate random numbers, and a way to ensure you seed differently on different processors
2. Track the total number of "live" (true, or 1) cells over time. There are 2 ways to do this - either adjust around the line which sets the true state, or count the cells after the update. What are the performance implications of these options? Is one or the other better? Does that depend on our initial grid?
3. Does parallelising the ix or the iy loop perform better? Why is this?

- 1C: Another Harder Program

- This time we have a 1-d array code which is filled with a sum over elements of a second 1-d array
- The idea is that we have a set of objects in a line which repel each other. Over time they should end up roughly evenly spread. For a bit of interest, we have a sort of "history" where previous positions influence future ones - the parameter "force_retention" controls how strongly this happens. For a value of 0, the elements should spread evenly and stay there - for a value near 1 they will overshoot and oscillate (wobble). This is mocking-up a electrostatic physics problem, but not actually coded correctly for one.
- Set the underlying array in the provided code to a sensible length (around 20), and set *show* to true to see the results on screen. For OMP to scale, you will need to increase this length to a few tens of thousands - do turn off the printing before doing this!
- This code file is called Repulsion.[f90|c]

Info:

Again, read over the sample code and make sure you understand it! Compile and run the code too. Note you will need to include "-lm" in the link step for the C code. Does the code do what you thought?

What parameters are there? Which might affect the parallel operation?

Problem Description:

There are various nested loops in this program. The time loop is NOT suitable for paralellising, because each iteration depends on the last. Which of the others can be paralellised? Which might be "high impact"?

Are there any places you could turn the code into a loop that can be paralellised (especially in the Fortran)

Try adding the simplest paralellism. Pay attention to any variables need to be private! Run this on a small grid and verify the answer is unchanged on 1, 4 and more (if possible) processors.

Check the scaling on larger problems. If it is poor, think about whether you parallelised the right loop!

Part 2 - Theory and Philosophy

- 2A: An easy example

- Rather than mechanically writing code and trying to bully it into working, it can be good to deliberately not write parallel code and instead just plan how it would be done
- In this example, we're going to describe how we'd paralellise, but the idea is to work this out without trial and error

Info:

Once you've got the plan, the best verification is to code it up and test it. Some of the things to test are:

Run on several processors, and then a different number. Are the results the same?

Run on a single processor. Does it still work? Some things can deadlock here if you weren't careful in the design

Run on say 2 and 4 processors. Is there a speedup? Is it as expected?

Test a range of input values, and verify that all get sensible results

Problem Description:

Suppose you have a large 1-D array full of values, and you want to compute their statistics (average and standard deviation). How could this be paralellised?

There's often more than one approach, and there definitely is here. If you can think of more than one approach, consider the relative efficiency and clarity of the options. (Note - you can make guesses about efficiency, but can only really tell by testing).

- 2B: An harder example

- This example is just like the previous - we don't want to write code, we want to plan it! Then we can write it out and test it to verify our ideas

Info:

Remember to check the same things as in 2A!

Problem Description:

THIS IS NOT A SENSIBLE ALGORITHM - this is meant to illustrate paralellism. If you can't imagine what this might be used to actually do, not can we...

Suppose you have an array filled with random, positive, non-zero, real (decimal) numbers. You want to do the following:

1. Calculate the average of each row in the grid
2. Calculate the average of all rows (average of averages of rows)
3. Take the row with the largest average, then replace the largest value in it with the global average
4. Repeat until the largest row average is within a fixed factor of the overall average. If/when you code this up, try a factor of 2.

This is some kind of a relaxation process, but not a terribly meaningful one.

What loops are involved? Write out a flowchart or model.

What can be parallelised? What if the grid is not square? How can you use processors efficiently?

What things need to be reduced (shared) or summed?

Extension - if programming this "for real" (for a working project), an important efficiency saving would be not recalculating needlessly. What things could yo preserve over iterations? How? How would you manage the parallelism?: