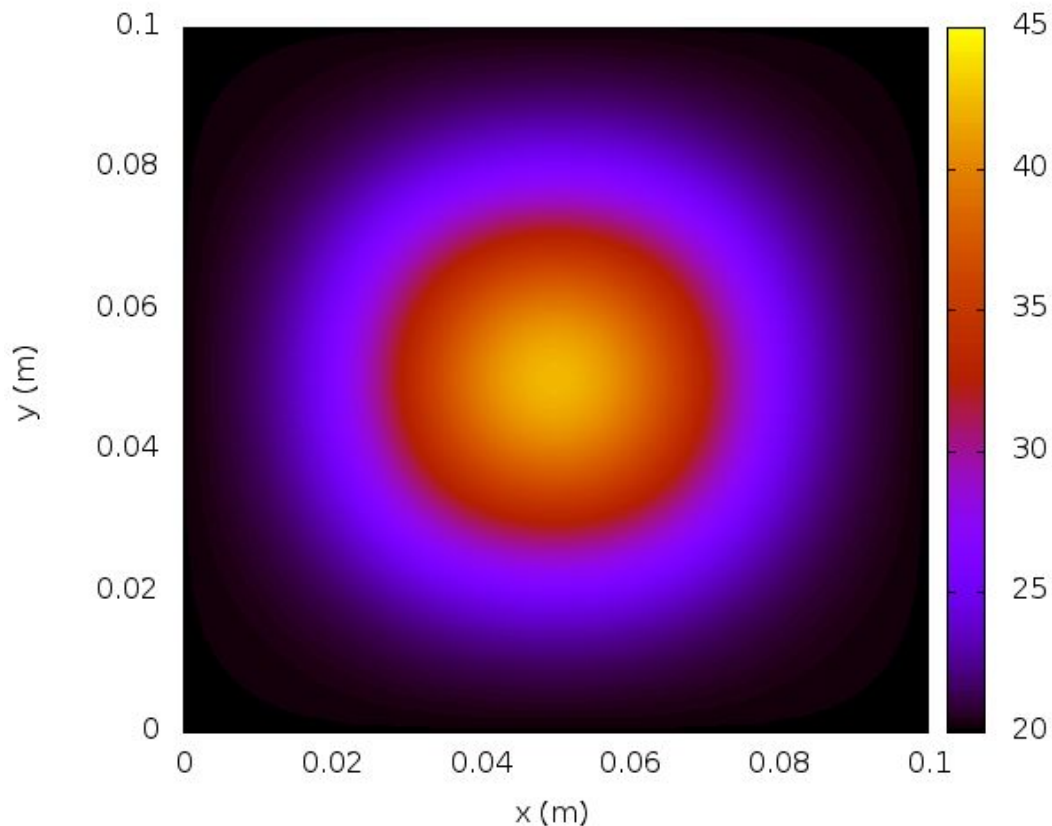


ECM 3426 Continuous Assessment 1

Heat Transfer Equation Parallel Scaling Study

Once I had written a serial C program to implement the given heat transfer equation I had to ensure that the results matched what was expected. After I was happy with my results I was able to move on to parallelising the program with OpenMP.



There are two distinct for loops in my program, provided you aren't writing out the results to a file. One for initialising the 2D array of initial conditions and the other for actually implementing the equation and updating the array. The second loop also contained four more nested for loops. This worked out to have the outer loop iterating the time step, the second-most inner loop iterating i (the x-axis) and within that j was being iterated (the y-axis) and the calculation was carried out, repeating this again to update the values.

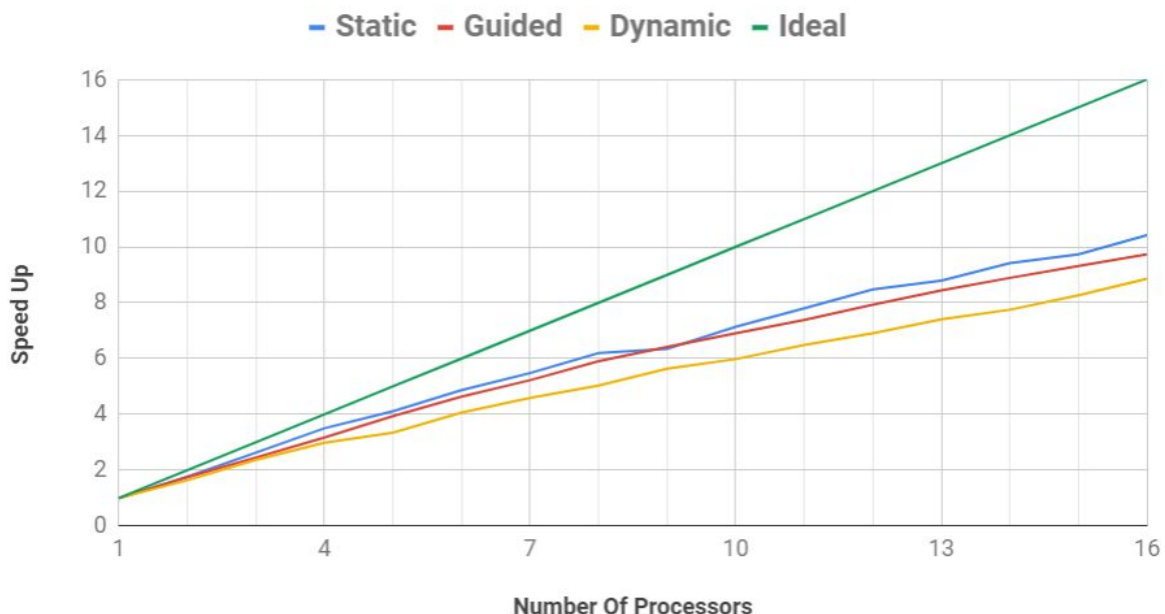
By simply placing print statements before the execution of each loop I could crudely observe the time taken for them to execute. It became clear that the large majority of the work was taking place in the second loop. Initialising the array was almost instant.

Despite knowing that it was likely unnecessary in terms of improving performance, my first step was to parallelise the first loop. This was partly to make sure I was implementing OpenMP correctly before continuing, but also to confirm my assumption that speeding up the creation of the array was insignificant. This proved to be the case with the execution time remaining around 24 seconds for running the program in serial and parallel. I left the scheduling as static as no part of the loop could require more work to be done than any other.

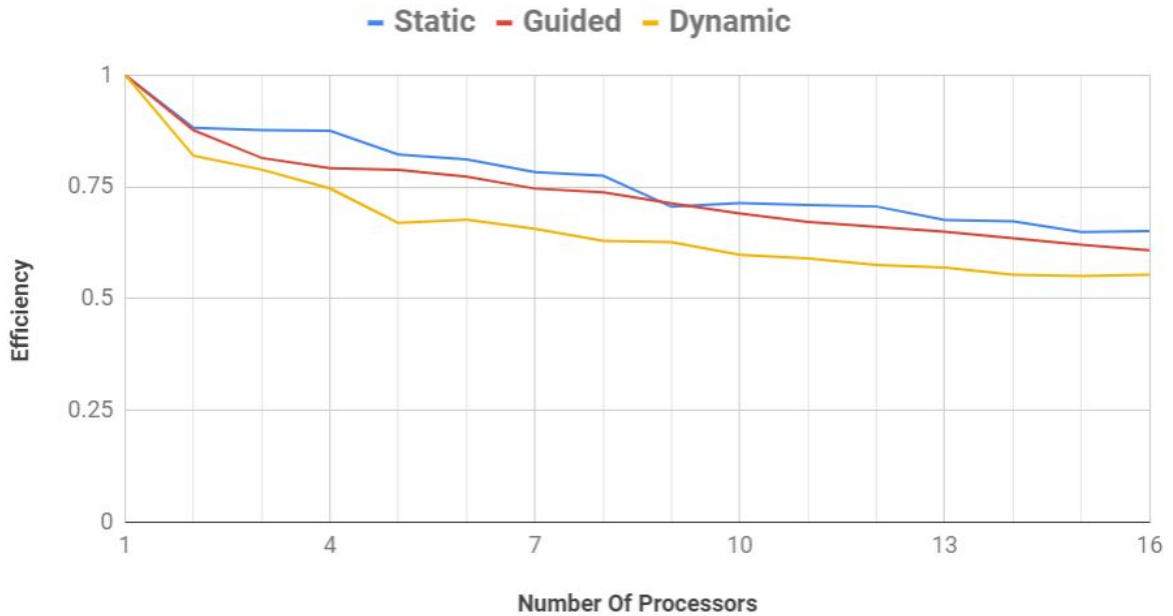
For the next set of nested loops, I had to decide which loops to parallelise. It could not be the outer one iterating the time steps as each iteration is reliant on the results of the last. Therefore I opted to parallelise the two second-most inner loops. I then checked that the results were still correct when the program was run in parallel before changing the program to no longer do any form of IO. This was to ensure I was just measuring the impacts of parallelising the computation of the equation as writing to the file added a significant amount to the execution time.

The next step was then to investigate the effects of different scheduling modes for the second loop and how they scale with more processing cores. All of my results are based on a serial execution time of 23.56 seconds. My results show that static was the best performing mode, maintaining an efficiency of above 0.65 even at 16 processing cores. Guided also performed well, with it and static retaining a better speed-up until more than 4 cores. After this point their efficiency began to dip more strongly and their speed-ups diverged from each other.

Speed Up Of Different OpenMP Scheduling



Efficiency Of Different OpenMP Scheduling



The results from dynamic scheduling were the most of note. The efficiency immediately dipped the fastest, reaching 0.75 at 4 cores, with it plateauing around 0.55 after 13 cores. Evidently dynamic performed much worse than the other two and at least one factor was causing this.

Dynamic scheduling is suitable when each iteration might take a vastly different length of time to execute. By dynamically allocating the work, the loop can be more efficiently load balanced. For the calculating the heat equation shown below, the amount of work needed to be done for each iteration essentially does not change.

$$\frac{\partial T}{\partial t} = \alpha^2 \left(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} \right)$$

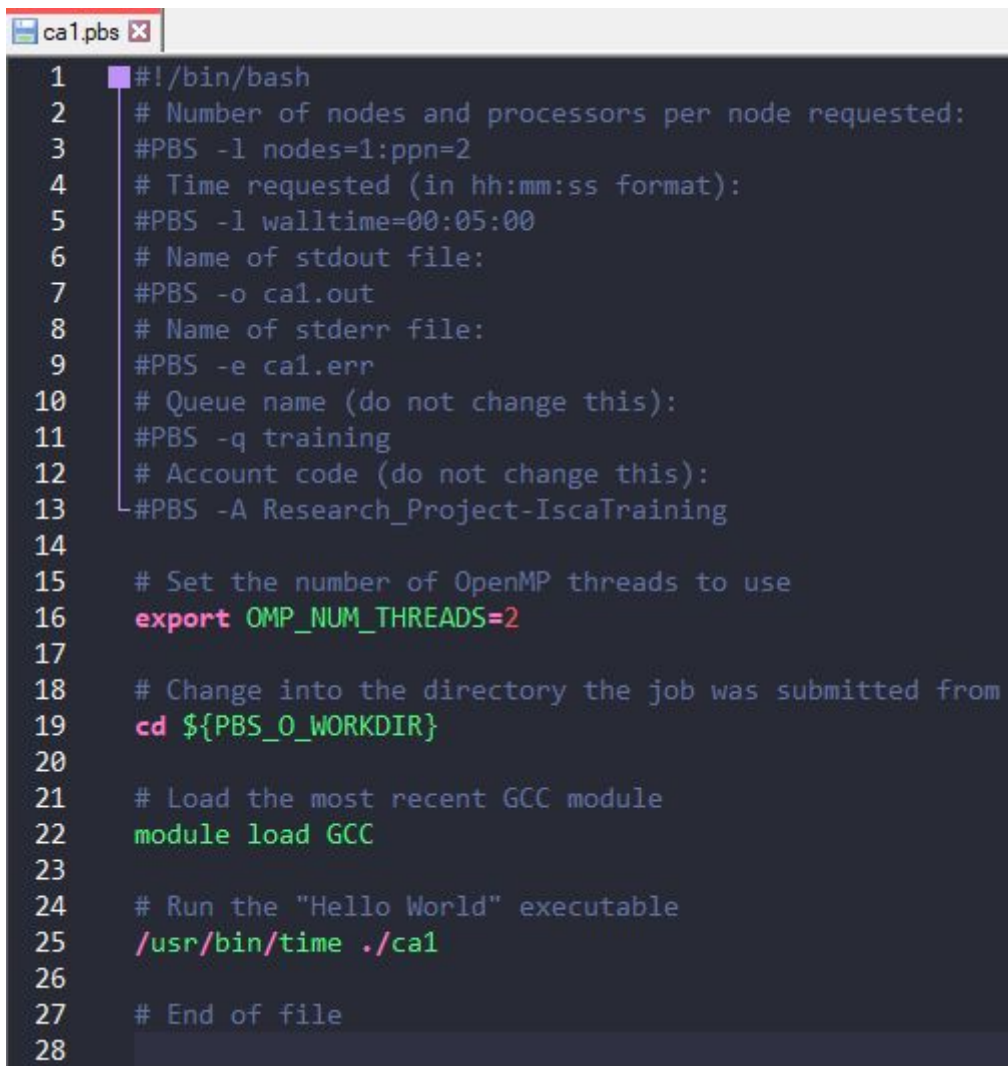
This fact allows static scheduling to still be properly load balanced so that some threads are not waiting for others, whilst also benefiting from its advantages over dynamic. First of all, there is less overhead in terms of managing a statically scheduled loop than a dynamic one. This leaves less time wasted on allocating the work to each thread and therefore a shorter execution.

The other, perhaps overlooked advantage of static scheduling, is how it handles NUMA systems. As dynamic has to dynamically allocate work to each core/thread, the values that that core needs to access could be stored

anywhere in memory as it cannot be predicted which core will be 'ready' for more work first. This will result in lots of cores having to access data from outside of their NUMA node, potentially drastically worsening performance. With static scheduling and each thread having a predetermined block of work to do, it will only ever need to access memory on its own NUMA node and this location will not change for each core.

Instructions:

In order to repeat these tests you will have to edit two files, `ca1.c` which is included in this submission, and a `.pbs` file that structured similarly to the image below. First of all, make sure that `doIO` on line 16 is set to 'false'. This ensures that the program does not output to a file. This is important as it can affect the execution time which we are trying to measure. Then you will need to set the scheduling of the loops on line 44 and 50 (leave the first loop on line 28 untouched). In the brackets after the word `schedule`, enter in the scheduling mode you wish to test the scaling of.



```
1  #!/bin/bash
2  # Number of nodes and processors per node requested:
3  #PBS -l nodes=1:ppn=2
4  # Time requested (in hh:mm:ss format):
5  #PBS -l walltime=00:05:00
6  # Name of stdout file:
7  #PBS -o ca1.out
8  # Name of stderr file:
9  #PBS -e ca1.err
10 # Queue name (do not change this):
11 #PBS -q training
12 # Account code (do not change this):
13 #PBS -A Research_Project-IscaTraining
14
15 # Set the number of OpenMP threads to use
16 export OMP_NUM_THREADS=2
17
18 # Change into the directory the job was submitted from
19 cd ${PBS_O_WORKDIR}
20
21 # Load the most recent GCC module
22 module load GCC
23
24 # Run the "Hello World" executable
25 /usr/bin/time ./ca1
26
27 # End of file
28
```

You will then need to compile `cal.c` without OpenMP to measure the time taken when the program is run in serial. Type **`gcc -o cal cal.c`** to compile. You will then need to type **`msub cal.pbs`** to submit the work to the queue. Once the job is complete, **`cal.err`** should contain the time elapsed. This is your value for the $N = 1$, or running in serial.

You will then need to compile it again with OpenMP using **`gcc -o cal -fopenmp cal.c`** and edit the `.pbs` file. To run the job on two cores, line 3 should read **`#PBS -l nodes=1:ppn=2`** and line 16 **`export OMP_NUM_THREADS=2`**. Submit the work again, and read the error file for the time elapsed. Repeat this with increasing number of cores up to 16.

To investigate a different scheduling method, repeat this process after changing the scheduling method on line 44 in `cal.c`.