MATH96012 Project 4

Matthew Cowley 01059624

December 13, 2019

Part 1 -Blood flow through a deformed artery

1.1 A is very sparse matrix so matrix multiplication is very inefficient. So instead you make use of the features of A i.e it is basically tridiagonal with 2 additional diagonals if you ignore entries due to boundary conditions. So when working out z=Ay you obtain z(i) by combining certain values of y. This means instead of doing $(N+2)^3$ calculations you do roughly $5(N+2)^2$. Doing it this way means it is relatively easy to parallelize with open omp.

1.2 I calculated A^t in a similar way to 1.1, but far more complicated.

1.3 I parallelized mtvec and mvec as it iterates through the n (n+2)*(n+2) sub-matrices, I then parallelized using open mpi reduce where ever a dot product was needed. I also created as few variables as possible when implementing the sgi method.

Part 1.4-Performance

pls note there is a bug in the code so jocobi correct array is not outputed hence jacobi is not showing on the graph.

From figure 1 we see that the python Jacobi seems to perform the best, which is a bit unexpected as we would expect a compile version to be quicker. This is maybe due to the way python stores the information. We can also observe that that parallelising the code does not really impact performance, if anything it appears to reduce the performance. Obviously run time increase as n gets bigger as well.

As we can see from figure 2 there is no speed up, even for increasing m which is a little surprising. But this is mainly due to the way the code has been parallelised, as it calls too many parallel regions every iteration. It is also due to the percentage of code which has been parallelised and that is not much

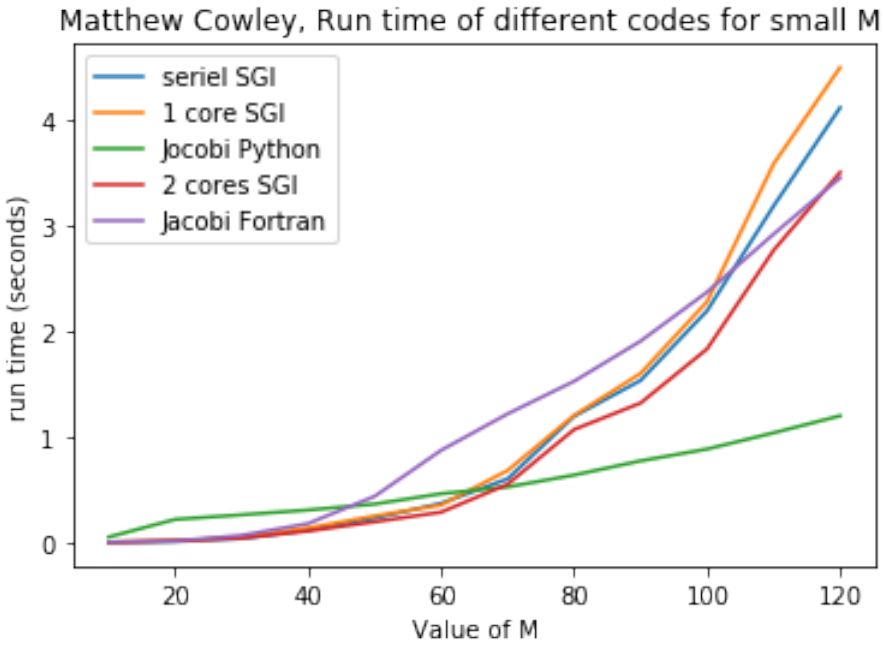

Figure 1: How varying codes and n effects performance

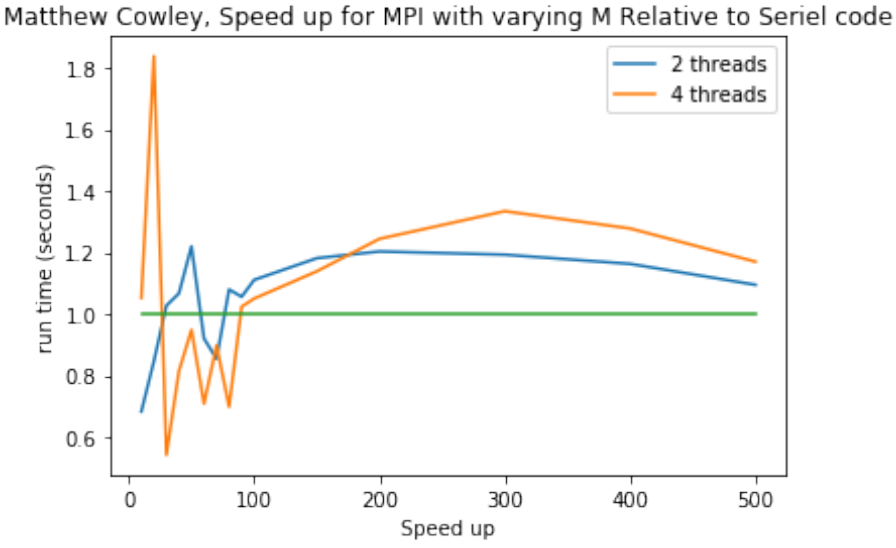

Figure 2: How effective MPI is for more threads

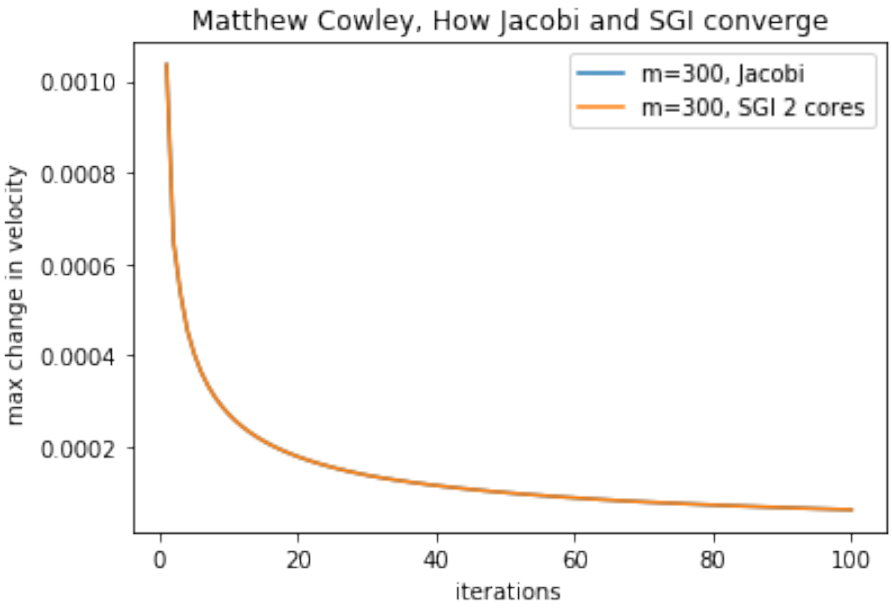

Figure 3: convergence

really. Also there are far to many bottle necks really. For example if we look at Amdahl's law,

From figure 3 we can observe that Jocobi method isn't actually giving us the right answer and is converging much slower, especially for larger n. This is expected due to the nature of the methods. This explains why in figure 1 it appears that Jocobi seems to be faster, but it actually has a lower rate of change due to poor convergence, so appear to have converged. Lowering the tolerance would prehaps have solve this.

Part 1.5-MPI Plan

First of all you set up your MPI region in the start of sgisolve subrountine and use MPI library, then call MPI INIT, MPI COMM SIZE ect.. finishing with MPI FINALISE at the bottom of the subrountine.
I would start by not having mvec and mtvec, but split them across processors. I would split these calculations $(n+2)^2$ into the number of processors using 1DCOMPGRID, assigning each processor one part of the calculations, similarly the variables e.g x old,x new,d,e should be distributed across the processors with same indexing i.e $z_i...z_j$ and $d_i.....d_j$
You would keep the the sum variables used in code for part 1.3 (where reduction has been used e.g Ad sum,e newsum,e oldsum plus x diffsum in this case) on processor 0, making other processors communicate their value using ISEND and receive them using GATHER on myid=0. you then use these to calculate scalars like κ,μ , you then send this information to all the processors so they can calculated next step in sgi method use ISEND and RECV. Each processor stores its on part of the updated sum values.
At the end you combine the x new onto processor of myid=0 using GATHERV and ISEND on all processors.
This method would be quicker as you would not be calling multiple parallel regions every time steps and sharing the computation load more efficiently across the processors. It could also handle a larger N as the variables are spread across the processors and only x is gathered onto one processor at the end. However there are probably better methods to solve the problem and better suited to using MPI and being parallelised e.g maybe multi-grid. A con of this method is that is would be very fiddly to code and not good for small N. The main pro is that you could then uses many cores to solve the problem like Imperial cloud service, instead of a shared memory computer, thus solve a much bigger problem.

Part 2 -Weakly coupled oscillators on a simple network

2.0- You have to compute theta at every time step for all the N oscillators (locusts in this case, a they are being modeled as a swarm and N is supposed to be large). This $\theta_i(t+\delta t)$ depends on the previous time step $\theta_i(t)$ and other N's $\theta_j(t)$, the amount that of other $\theta_j(t)$ that effect $\theta_i(t+\delta t)$ is dependent on an assigned value of a_i e.g for $a_i = 1$ only the neighbors either side will effects it. We are assuming in this case that the oscillators are modeled as being in a circle (in reality they won't,which is what model in 2.4 addresses).

In short you split up which theta's you are solving (e.g θ_1 to $\theta_5 0$ and assign them to a core for it to solve. However some θ_i's need θ_j's from other core's to

be computed at each time step. So at each time step you send θ_js needed by the other core's to them and likewise receive them for each core. The amount of theta's you need stays constant, as it depends on a_i which is fixed for all time.

So my code starts by finding the minimum amount of θ's each processor needs from other processors, by analysing a_i and it's closeness to the processors θ's boundary's. It does this using MPI isend and MPI recv. It then extracts the relevant initial conditions needed for each processor.

Then as it progresses through time each processor sends and receives the θ's it needs calculating the next time step using the subroutine RHS and then using an explicit Euler-method to advance the θ's value in time.

The code then calculates r the degree of synchronisation for a given time step storing the information on processor 0. Along with at the end producing a θ value for every oscillator on processor 0 for a final given point in time.

2.1 I used ISEND to send to the other cores the dimensions they are expecting (max a_i relative to its index). and RECV to receive them (it acts as barrier meaning no core ever really gets ahead). After this each core sends relevant θ to other cores every time step.

2.2 I used ISEND and RECV instead of GATHER, since GATHER was having issues receiving complex numbers. This method means it is not quite as efficient as it could be, but the loss in efficiency is very minor.

2.3 First I sent the dimensions to be expected for all the local ys from all the processors to processor 0 (using ISEND and RECV), after this each core then sends it's y local at the end of simulate. Processor 0 gathers them by using gatherv.

Part 2.4

We would need to change the dimensions of various variables substantially either making them far larger or 2 dimensional. For simplicity lets say larger. We would then have to write a code similar to the previous problem apart from you have to send and receive information from more processors i.e visually you know have above, below, left and right provided M¿¿Numprocs. So you would need more ai's max information communicated and more information like start and end points of *theta* data, as they would no longer be at ends of farray but in the middle as well. You would do this using ISEND and RECV. A way potentially reduce communication could get round this is to divide you domain into strips, so you only have transfer data to processors above or below. However this is

more effected by M large compared to splitting your grid like a window frame . If a_i has high variance this also causes issues with both grids as you have transfer far more data. It would be very fiddly to modify, but would see big improvements for far more processors, e.g the imperial computing cloud service would be useful in this case.