Parallel Simulations of Gravitational Interactions: a Direct Approach

H.H. Wills Physics Laboratory, University of Bristol, Tyndall Avenue, Bristol BS8 1TL, UK

8 February 2017

ABSTRACT
The simulation of n-body systems is a perfect situation for exploring parallel computation. The key ideas of parallel computing will be explained along with how these techniques are applied to these simulations. A single core code has also been used to measure parallel code improvements. This report presents efficiency data for a variety of initial parameters. Two different approaches to parallel computing are explored, a shared and a distributed memory method.

1 INTRODUCTION

Modern computers have processors with multiple cores. This opens up an opportunity to run some calculations over multiple parallel threads, instead of one (serial) Grama (2003). It is important to appreciate the structure of a processor in order to understand how parallel code runs. Cores have their own small amount of cache memory, the fastest type of memory. This memory however, can only be accessed by that one core. Zoom out a layer and each core in a processor is connected to shared layer of cache, data can be shared between cores in this memory. Clusters are a collection of individual processors that are linked such that they can quickly communicate (low latency and high bandwidth). Parallel code is specifically designed to spread calculations and take advantage of a large number of cores.

In parallel it is often the case that calculations require pieces of information previously calculated over many cores (data dependencies), this information must be communicated between cores before the dependent calculation is executed Andrews & Schneider (1983). Understanding which parts of a calculation can be completed with and without communication is key, as communication is a major overhead (any time code spends doing anything other than calculation).

There are limits on the potential speed up even well designed parallel computation can provide. There will be always parts of a program which are not parallelizable and act as a bottleneck. Amdahl's Law states that ultimately any speed up obtained using parallel techniques will be limited by the fraction of code which is not parallelizable Pacheco (1997).

2 SIMULATIONS

The simulations begin with definitions of initial positions and velocities of n point-like particles, generated on a master core before entering the main loop. The code then splits into parallel to calculate the forces between each particle pair using the following equation;

$$F_{i,j} = \frac{gM_iM_j}{|r|^2} \tag{1}$$

where $F_{i,j}$ is the force of particle i on particle j and g is the gravitational constant (can be set to 1) Aarseth & Aarseth (2003). These forces are then separated into x and y components using the angles between the current positions of each particle, in this step the sign of the component is also corrected based on current positions. The number of calculations required is reduced in two ways. Firstly, if the force data is stored in an matrix F any $F_{x,x}$ element (diagonal) equals 0, so isn't calculated. Secondly Newton's third law states elements in this matrix obey the following,

$$F_{i,j} = -F_{j,i} \tag{2}$$

this allows us to fill the whole matrix by only calculating the top half. (The calculated information must then be synchronized on the master core. The next steps have dependencies on the results, currently spread across the memory of the slave cores.)

Components are summed to get the overall force on each particle which is fed into the following equations of motion;

$$v'_x = v_x + \frac{F_x}{M}t \tag{3}$$

$$x' = x + v_xt + \frac{1}{2}\frac{F_x}{M}t^2 \tag{4}$$

where v' and x' are the new velocity and position respectively and t is the time-step value which is defined at the start of the program. Each loop of the code simulates a length of time equal to one time-step therefore the smaller the time-step the more accurate the results, but the more computation required.

The Newtonian potential this creates has a $1/r$ singularity for point-like masses Barnes (2012). Particles attract each other, as they get closer the forces between them increase. There always reaches a point where the acceleration the forces provide is large enough to, in a single time-step, speed the particle up enough to 'jump' to a new position beyond the attracting particle. The velocity gained is then over the escape velocity of the force from the attracting particle pulling back. This means instead of naturally colliding or entering orbit like behaviours, particles effectively repel each other at extremely short distances. To combat this an approximation must be made, this is the softening parameter Trenti & Hut (2008) which modifies the force equation to;

$$F_{i,j} = \frac{gM_iM_j}{|r|^2 + \epsilon^2} \tag{5}$$

where ϵ is the softening parameter. This is a cushion distance where the strength of gravitational forces are reduced. The value for ϵ is situational, ϵ must be small enough not to affect long range interactions (this is the case where $r \gg \epsilon$) but large enough to correct the forces in short range interactions. The optimum value for ϵ will depend on the speed of particles when approaching collisions, an approximation can be made from the original mean distance between particles Rodionov & Sotnikova (2005). When simulating the dynamics of planetary orbitals the softening parameter is not necessary as objects in stable orbits do not collide.

The simulations are computed in a single 2D plane for simplicity (although 3D code was written see FIG. 2) and allowing for simple graphics.

Initial conditions have been set for particles in a cloud and objects in the Solar System. Stability of these systems depends on the time-step used in the simulation and distance and mass parameters involved. Stability and the impact of the softening parameter can be observed by tracking the centre of mass of a cloud of particles. For stable and accurate systems where the softening parameter has little impact there should little to no movement, which is observed when the time-step is small enough (example values of this time-step are again, situation dependent).

2.1 Distributed Memory

MPI (Message Passing Interface) is a language developed for controlling the running of code over multiple cores which do not have access to any shared memory, for example nodes in a cluster. This is a massive advantage of MPI, it is scalable in systems with huge numbers of cores. MPI runs the same code on every core, this is then split up based on the rank of each core.

The code assigns a master core to collect, sum and update the data. All cores (including the master) compute a share of the force calculation. Load balancing between cores in MPI code must be specified, it isn't automated. The approach taken here is simply to divide load equally (by the number of calculations for simplicity) and have the same cores compute the same calculations every iteration. This is achieved by first getting the total number of matrix elements which require calculation, m, in a system of n particles this is obtained using the following,

$$m = (n-1) + (n-2) + \ldots + 1 \tag{6}$$

this number is then used to distribute the workload between cores. This doesn't perfectly distribute load evenly for two reasons. Firstly a simplification is made in the code, any remaining elements when m is divided by the number of cores (p) are computed on the master. Secondly two cores doing the same number of calculations will not take the same time, it can depend on where data is temporarily cached.

Communication on MPI is necessary when sending data to and collecting results from slave cores. Communication is the major component of overheads in distributed memory code. Although the actual communication event can be fast in low latency systems, packaging the data ready for sending takes time and waiting for recipient cores to be ready can mean cores wait, wasting time.

Communication can be achieved sending single matrix elements one by one with MPI_Send() and MPI_Recv() commands but this is woefully inefficient. These commands are blocking (these functions do not complete until message is received), cores spend time waiting for a previous command to complete leading to the inefficiency. Non-blocking commands are available but there is risk of one command overwriting data transferred by a previous command.

A faster approach (50 times slower (but situation dependent) when tested) is collective communication, using MPI_Bcast() and MPI_Gather() which can send arrays in single commands. The MPI_Bcast() is a blocking communication algorithm which uses a tree implementation to cycle through sending array elements using multiple network links simultaneously, sending data to all cores from the master. MPI_Gather() then collects data from each core into a single array on the master Bruck et al. (1995). The MPI code run here has barrier synchronization, the calculations on all cores must complete before any communication of that data starts.

The MPI code written here is coarse in it's granularity, with relatively large chunks of calculation between communication events.

2.2 Shared Memory

OpenMP is a shared memory interface. It takes advantage of the layer of cache that is shared between cores. This gives OpenMP it's advantage over a distributed memory approach, it doesn't have to communicate. Although it does still has overheads with data access, as only one core can access a single piece of information at once. This leads to some cores waiting, but this overhead is much lower than that of communication in a distributed memory system.

OpenMP code runs as a single master thread with sections that split into parallel, these sections come back together into serial when all calculation is complete and the code continues. This introduces synchronization barriers which can only be crossed when all calculation is complete, this implies imbalanced load distribution will be a major source of overheads.

OpenMP allows the calculation load each core has to be dynamically set Chandra (2001), instead of being determined at the start as with MPI. Here the calculation load is balanced with dynamic scheduling which can be tuned to improve performance. The effects of data dependencies in calculations is not removed in shared memory due to synchronization barriers. Instead the improvement comes from the reduction in time a core may wait for others to complete calculation. Producing the same effect in a distributed memory approach would require too much communication and wouldn't increase performance.

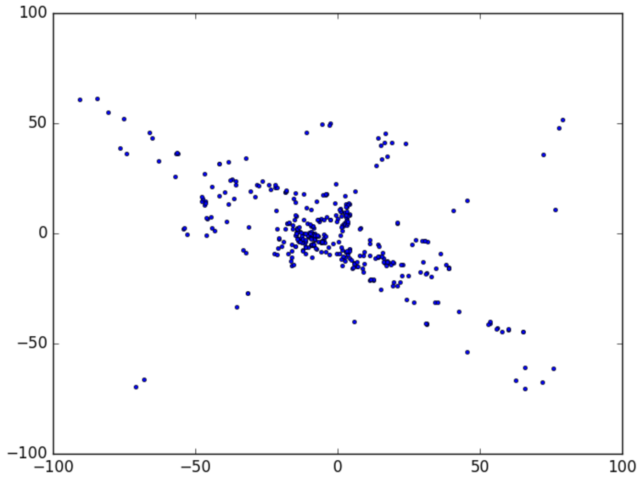

Figure 1. Graphics for a 400 body simulation, particles began stationary in randomized positions. Each particle is point-like with equal mass. Axes are arbitrary length units.

In a situation requiring less than 8 (16 on BlueCrystal) cores, OpenMP code is expected to be faster than MPI code. The disadvantage of OpenMP comes as it's limited to single processors, any large n system will be more efficient on a larger number of cores.

3 GRAPHICS

Although calculations for the simulations were coded in C, graphics for the results produced were written in Python separately using matplotlib. Data is read into Python from a data file created by the C calculation code. This data is then plotted with Pyplot in a graph which refreshes every time-step. The delay between plots can be tuned to produce an animation. FIG. 1 and FIG. 2 show single frames for animations for a group of particles in 2D and 3D.

4 COMPARING THE SIMULATIONS

Variations of these simulations for serial, MPI and OpenMP code were run and timed using the clock_t clock(void) function in the C standard library and omp_get_wtime() in the omp.h library. Data was collected from runs on BlueCrystal (16 x 2.6 GHz SandyBridge cores) and on an Intel(R) i7-6700HQ 2.6 GHz processor (i7 from here). Errors displayed in plots are the standard deviation in the mean result from 3 runs, errors arise from the interactions with background processes running on the same cores that will inherently disrupt the program to some extent and variations in where calculation data is temporarily stored.

We can define an ratio which will approximate the efficiency of a program by the following equation,

$$\eta = \frac{T_1}{T_N} \tag{7}$$

where T_N is the time taken to run with N cores and T_1 is the time taken on 1 core.

All times are plotted in arbitrary units, as for any two

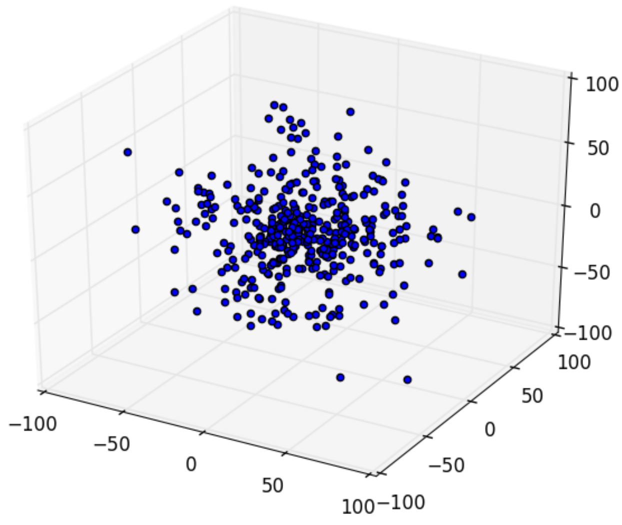

Figure 2. 3D version of the simulation. Graphics for a 400 body simulation, particles began in stationary in randomized positions. Each particle is point-like with equal mass. Axes are arbitrary length units.

runs with the same physical parameters (where the number of iterations $\gg 1$) any increase in the time taken is due to an increase in the number of iterations. Any selection of the number of iterations in a run is arbitrarily chosen so that each simulation runs in a reasonable time, however the time taken must still be long enough to be statistically significant.

4.1 Comparing Serial, MPI and OpenMP code

Over a small number of cores (less than 8 on an i7 or 16 on BlueCrystal) it is expected that a shared memory approach to the calculation will be most efficient. This was observed, the data in FIG. 3 shows the time the same program took to run for various n. This data was collected from runs on an i7 running all 8 cores.

For very small n systems (less than $n = 10$ for this system), running code in serial is faster. With such a small number of particles, there is very little calculation required and the time taken to communicate results is more than the time saved running calculations over distributed memory. This is an example of a system with extremely low granularity (the calculation chunks of code a much smaller that communication).

At this low n the shared memory approach is also slower. When code is run on one core data can be stored in the cores individual cache memory which has less latency than the shared memory bar across all cores ($\sim$ 10 CP (clock periods) compared to $\sim$ 300 CP). Low granularity and conflicting cores trying to access the same data also contribute to the slow down.

For the small range of cores plotted in FIG. 3 OpenMP was the faster than MPI, as expected. This is due to the smaller overheads compared too those in MPI communication and processors waiting due to bad load balancing. The improvement from better load balancing may be limited as this OpenMP data is from code with static scheduling, although even by default the scheduling OpenMP uses will

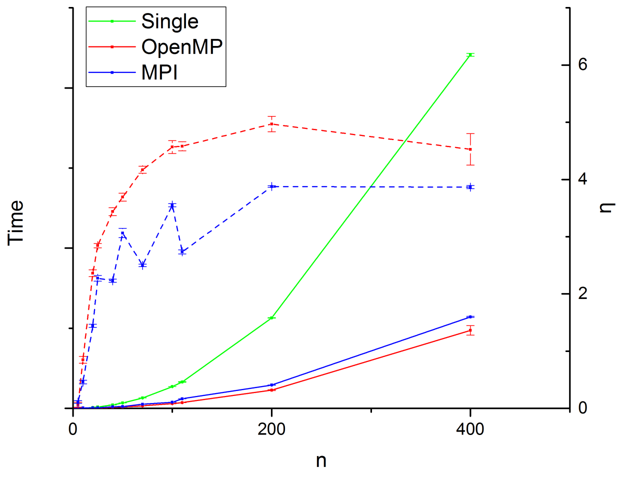

Figure 3. Comparing the performance of each approach over a range of n. Data was recorded on an i7. Here MPI and OpenMP code used all 8 cores. The OpenMP code here is using static scheduling. In the solid lines is plotted the time taken (arbitrary units) and in the dashed is the η value. The time data shows a clear n^2 relationship.

be more sophisticated than dumping the remainders on the master thread.

4.2 Scaling

In the region where $n \gg p$, increasing the number of cores generally leads to a faster program for the same n, see FIG. 4. MPI BC (run on BlueCrystal) and OpenMP show a similar behaviour as p increases. However this isn't true when an MPI program goes from 4 to 5 cores on the i7, this is artificial and is probably due to the characteristics of the i7's hyper threads interacting with MPI load imbalances and background processes on the cores. FIG. 4 also plots η which in this case shows the same relationship as inverse time because all three data sets displayed are from the same parameters and iterations. As mentioned, on a single processor where shared memory is available it is expected that OpenMP will be superior and it is.

As n scales we see time increasing with an n^2 relationship, see FIG. 3. This is due to the number of elements requiring calculation m increasing as,

$$m = -0.5n + 0.5n^2 \tag{8}$$

the same relationship is seen for single, OpenMP and MPI. η therefore levels off at a maximum value (see FIG. 3), which is expected if p remains the same.

FIG. 5 plots the scaling of η for MPI code as p scales over a range of values for n. Data was taken over a wide range of iterations for each situation. η is modulated by serial results with the same number of iterations and so is comparable. Comparing the η curves in FIG. 5 to those in FIG. 4 shows the same relationships. For a single n a program reaches a maximum η value as p increases. It is expected that when p is increased further there will a point where the extra communication requires too much time and it not worth the reduction in calculation time per slave core. At

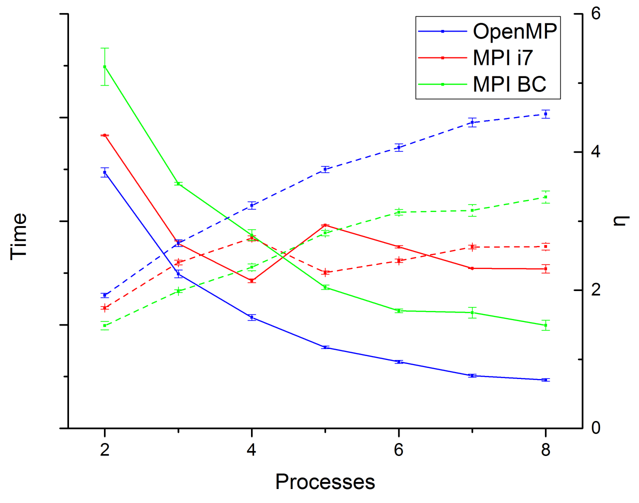

Figure 4. Plot to show the performance changes as the number of processes increases. OpenMP data is with static scheduling and is recorded on the i7. MPI data was recorded on both the i7 and BlueCrystal harware. Time is plotted with a solid line in arbitrary units and the efficiency η is plotted in the dashed lines.

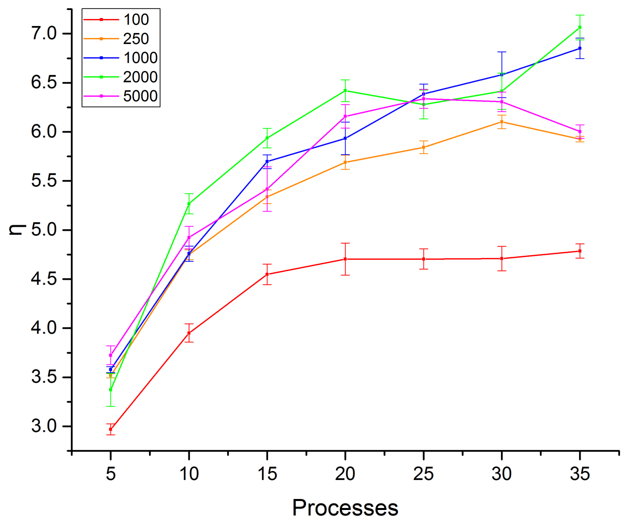

Figure 5. Plot showing the efficiency η data for MPI code as the number of processes scales. All data was recorded on BlueCrystal, across multiple nodes with 5 processes on each node.

this point η will reduce again. The peak η value corresponding to the optimum value for p, will be different for every situation, it will primarily depend on n but also on hardware specifications.

4.3 Tuning

As mentioned in the section on scaling, for a set n there will always be an optimum p. This value doesn't depend on the number of iterations so can be estimated through running low iteration runs before executing a long simulation at peak p.

For OpenMP simulations load balancing is, as dis-

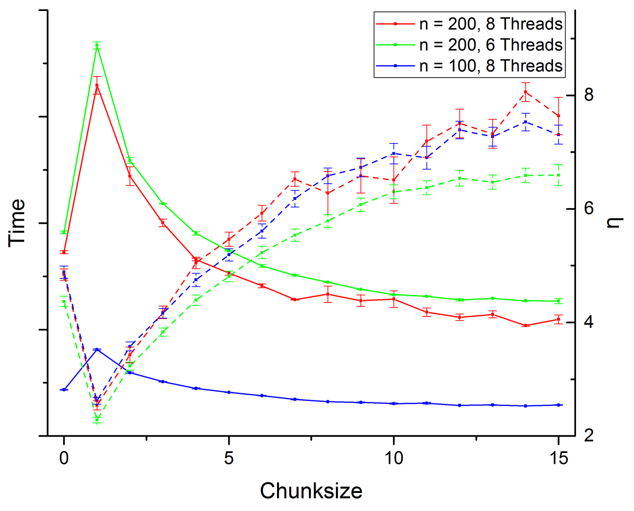

Figure 6. Plot showing the performance of OpenMP code as the chunk size c varies in dynamic scheduling. Time data is plotted with solid lines and efficiency η data in dashed lines. In this plot $c = 0$ represents static scheduling. The 3 situations plot here simply show that the relationship is similar across n and p whilst n is high enough, at low n when c is of the same order as n the effects may be more complex and the gain from dynamic scheduling may be non-existent.

cussed, a major source of overheads. Parallel loops in OpenMP offer static and dynamic scheduling. Static scheduling works similarly to the load balancing in this MPI code, a predetermined number of calculations are spread to each core. In this case if one core finishes early it must wait at a synchronization barrier for the other cores. Dynamic scheduling sends calculations to cores in chunks, when a processor finishes its calculations it gets the next chunk. This is more efficient as it reduces the impact of the synchronization barrier, instead of waiting for other processors a faster processor computes another chunk. The chunk size c can be set and tuned for optimum performance.

FIG. 6 plots the performance of OpenMP code over various chunk sizes. There is a large drop in η when $c = 1$ (maximum load balancing), dynamic scheduling continues to be slower than static until $c \simeq 4$. This may appear unexpected, the reason for this is that while chunks are this small the added computation threads must go through to distribute the small chunks takes longer than the gain equal load balancing provides. As c increases we approach an optimum value. Similarly to increasing p in FIG. 4 if c increases past the peak η will actually reduce again because the chunks become so large that overheads with load imbalance begin to appear again.

5 CONCLUSION

The efficiency (η) of both OpenMP and MPI code increases to a maximum value as n increases for constant p, with the position of this peak depending on p. Similarly with constant n, increasing p increases η again to a maximum value. Tuning the dynamic scheduling chunk size leads to an increase in η to a peak value, using chunks larger than this peak value leads to a decrease in η. Although not discussed here the calculations for efficiency per core can reveal diminishing returns for increasing p. In situations where the number of cores available is limited this may be relevant.

6 IMPROVEMENTS INTO THE FUTURE

The load balancing in this MPI code is currently simplified, computing all the remaining elements on the master. A more sophisticated solution would instead be to split the remainders up sending one to each processor. Kaehler presents methods for improved load balancing with MPI Kaehler (2016).

Currently all the equations of motion are computed on the master, results are then sent out to slaves for calculation. This potentially could be improved. After the master has summed the x and y components, these values could be sent to slaves for computation separately. This would require more communication but could provide a speed up. This is the approach of the OpenMP code.

The MPI_Reduce() function takes data from all cores similarly to MPI_Gather() but additionally it sums the values up. MPI_Reduce() is not used in this code however it could improve performance of the MPI code if implemented. With the complexity introduced through the load balancing currently in place, where forces on single particles are likely to be computer across multiple cores, MPI_Reduce() hasn't been added.

In OpenMP guided scheduling is a similar approach to distributing load imbalance as is dynamic scheduling. Guided scheduling however reduces c as the program progresses, this removes the inefficiencies brought in by large c values in dynamic scheduling and will provide a speed up.

There are improvements possible with combining MPI and OpenMP code into a hybrid approach. With MPI distributing onto nodes which each run OpenMP code Jost et al. (2003). Current research is exploring the use of GPU (graphics processing units) clusters to create a parallel environment capable of a larger number of smaller calculations Huang et al. (2016).

It is worth mentioning the existence of non-direct algorithms that use approximations to acquire speed ups, he most popular being the Barnes-Hut method Barnes & Hut (1986).

REFERENCES

Aarseth S. J., Aarseth S. J., 2003, Gravitational N-body simulations: tools and algorithms. Cambridge University Press
Andrews G. R., Schneider F. B., 1983, ACM Computing Surveys (CSUR), 15, 3
Barnes J. E., 2012, Monthly Notices of the Royal Astronomical Society, 425, 1104
Barnes J., Hut P., 1986, nature, 324, 446
Bruck J., Dolev D., Ho C.-T., Roşu M.-C., Strong R., 1995, in Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures. pp 64–73
Chandra R., 2001, Parallel programming in OpenMP. Morgan kaufmann
Grama A., 2003, Introduction to parallel computing. Pearson Education

Huang S.-Y., Spurzem R., Berczik P., 2016, Research in Astronomy and Astrophysics, 16, 011
Jost G., Jin H.-Q., anMey D., Hatay F. F., 2003
Kaehler R., 2016, arXiv preprint arXiv:1612.09491
Pacheco P. S., 1997, Parallel programming with MPI. Morgan Kaufmann
Rodionov S., Sotnikova N. Y., 2005, Astronomy Reports, 49, 470
Trenti M., Hut P., 2008, Scholarpedia, 3, 3930

This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol - http://www.bris.ac.uk/acrc/.