Parallel Computation and Distributed Systems Coursework

Samuel Lord, Undergraduate, School of Computing and Mathematics, University of Plymouth
samuel.lord@students.plymouth.ac.uk

Introduction

The present study examines the advantages and disadvantages of parallelisation for an air-motion simulation as a dynamical system. Air motion is a key factor in weather prediction and simulation due to the fact that it affects almost all weather systems; temperature, humidity, cloud cover, etc. However, due to time and hardware limits, the present study utilises a highly simplified system; using Newton's mechanical equations and the pressure gradient force. In order to test the system thoroughly, an atmosphere will be generated (with a fixed seed for repeatability) utilising pseudo-random mass values to unevenly distribute air across the system, thereby constructing a semi-realistic initial state from which to simulate air motion.

Sir Isaac Newton's three laws of motion are fundamental to the development of the mathematical description of air movement. Specifically the second law of motion, in which force, *F* is stated to be the product of mass, *m*, and acceleration, *a* (Atkinson, 1981). This is used to derive the pressure gradient force, which describes the motion of air from high to low pressure (Atkinson, 1981)(Holton, 1992).

$$F = ma \tag{1.1}$$

$$-\frac{1}{\rho}\frac{\Delta p}{\Delta x} = \frac{\Delta v}{\Delta t} = a \tag{1.2}$$

Where ρ is the mean air density (mass per unit volume) of the air between the air position and the position at the center of the low pressure, *v* is velocity, *p* is pressure, *x* is the distance from the mass of air to the pressure centre and *t* is the time over which the change in velocity is being calculated, in this instance the simulation timestep. From the acceleration, we can calculate the velocity at which the air is moving at the end of the timestep, using the equations of motion (Hanrahan and Porkess, 2014).

$$v = u + at \tag{1.3}$$

$$v = u + \frac{-1}{\rho}\frac{\Delta p}{\Delta x}t \tag{1.4}$$

Where *u* is the initial velocity of the air and *v* is the velocity at the end of the timestep. Further, it is also possible to calculate the displacement of the air in the timestep, *S*.

$$S = ut + \frac{1}{2}at^2 \tag{1.5}$$

$$S = ut + \frac{1}{2}\frac{-1}{\rho}\frac{\Delta p}{\Delta x}t^2 \tag{1.6}$$

As air moves within the system, the pressure will also change. As the mass within a sub-volume decreases, the density of air within that subvolume will decrease (Zdunkowski and Bott, 2004).

$$\rho = \frac{m}{V} \tag{1.7}$$

Using the new density, it is possible to recalculate the pressure for the subvolume (Atkinson, 1981).

$$P = R\rho T \tag{1.8}$$

Where T is temperature, which in this system will be fixed at 292 K (19 degrees celsius) and R is the specific gas constant (287.058 J $kg^{-1}K^{-1}$ for dry air (Atkinson, 1981)).
At this stage, the system is in a state at which mass can be transferred between the subvolumes. In this instance, the proportion of air transferred will be calculated by using the displacement of the air.

$$r = S/W \tag{1.9}$$

Where W is the width of each subvolume and r is the ratio of the displacement of the air to the width. The mass of air transferred, m_t, is therefore:

$$m_t = r\,m \tag{1.10}$$

The process may now repeat to simulate the movement of air over longer periods of time.

Implementation

In designing the algorithms used in the CUDA and MPI implementations, Foster's Design Methodology was employed. The extent to which the algorithms comply with this methodology is outlined below.

Partitioning

The present study parallelises a simplified representation of air movement. Through domain decomposition it is possible to reduce the problem into smaller primitive tasks. In this case, a 3D decomposition results in the the volume of air being subdivided into sub-volumes which can be treated as a single block of air in the dynamical system. This allows for the parallelisation of the task as each thread in the implementation can deal with a single primitive task.
The primitive task for each sub-volume is the equations explained in the introduction, whereby each element may calculate its state after a given time period, the information is shared to re-evaluate the state of the entire system and then the process is repeated as necessary.

Table 1: Algorithmic breakdown in relation to Foster's partitioning criteria

Criteria	**Criteria Met?**
The partition defines (at least) an order of magnitude more tasks than processors in the target computer	Yes
Redundant computations and data storage avoided as much as possible	Yes
Primitive tasks are roughly the same size	Yes
Number of tasks is an increasing function of the problem size	Yes
Several alternative partitions identified	Yes

Table 1 shows to what extent Foster's checklist for the partitioning element of the design process was met.

Due to the nature of the problem, the number of tasks can be set arbitrarily. This is because the volume of air being dealt with by each task can be decreased to increase the accuracy of the system. For example, if the problem has to deal with a 100 km^3 volume of air, assigning each task a 50 km^3 volume would result in only two tasks, but a highly inaccurate simulation. Equally, subdividing the volume into 0.01 km^3 areas would result in 10,000 tasks, but far greater accuracy. Therefore, the tasks can exceed the number of processors by at least one order of magnitude, and in fact in so doing increases the accuracy of the system.

Redundant computations have been eliminated through the reduction of the problem to primitive tasks, allowing each parallel unit perform only the computations, and store the data relevant to the subvolume it is managing. The primitive tasks vary only in the numbers on which the calculations are performed, and are therefore virtually identical at all timesteps for all tasks.

Alternative partitions have been considered in grouping small numbers of sub-volumes per primitive tasks however this reduces parallelisation and is unlikely to offer any significant performance increase and so the chosen partitioning was selected.

Functional decomposition has also been considered. The primitive task for each sub-volume requires the following steps, also shown in fig. 1.

1. Calculate pressure differences to neighbours to find along what vector the lower pressure is
2. From the pressure differences calculate the acceleration of the air mass
3. Using the current speed, acceleration and time step, calculate the new speed of the air mass
4. Calculate the displacement in this time step.
5. Based on the displacement, transfer a proportion of the mass to the neighbour in the direction of the pressure vector.
6. Using the known volume of the sub-volume and the newly calculated mass, recalculate the density of the air in the volume and therefore the pressure.

Due to the fact that each step requires the previous step to have completed prior to the next having the information required to execute, it is not possible to further functionally decomposition the tasks.

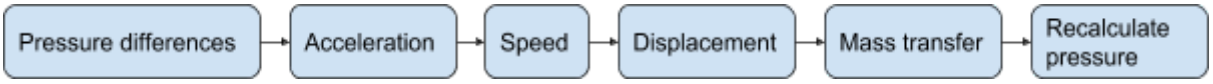

Figure 1: Air motion algorithm flow chart

Communication

Table 2: Algorithmic breakdown in relation to Foster's communication criteria

Criteria	**Criteria Met?**
All tasks perform the same number of communications	Yes
Each tasks only communicates with a small number of neighbours	Yes
Communication operations can proceed concurrently	~Yes
Tasks can perform their computations concurrently	Yes

Table 2 shows to what extent Foster's checklist for the communication element of the design process was met.
Each task performs an identical number of communication to all others. This is due to the fact that the simulated atmosphere is treated as a taurus where the edges of the three-dimensional grid wrap to the opposite side in a similar manner to the real atmosphere. This results in each task only communicating with a small number of neighbours (six). This occurs twice in each timestep, firstly to get the pressure of surrounding sub-volumes, and secondly to transfer the mass of air between subvolumes. The communication performed by each task can proceed concurrently because a given subvolume can asynchronously send and receive pressure and mass data for all neighbours simultaneously. However, a task must wait until it has received data from all neighbours to continue execution. Further, computations can be performed concurrently by all tasks until communication must occur.

Agglomeration

The implementation presented has not been agglomerated to high degree. This is due to the fact that the primitive tasks created through domain decomposition are well suited to parallelisation. An alternative agglomeration was considered in which tasks would be made up of a small number of subvolumes (which may scale with problem size) and information would be collated within the group and then be disseminated to other groups. However, the alternative implementation would suffer the same shortcomings (in serialisation bottlenecks,

fig. 2) as the selected implementation, with added complexity in implementation requirements, and so was discounted as a viable option.

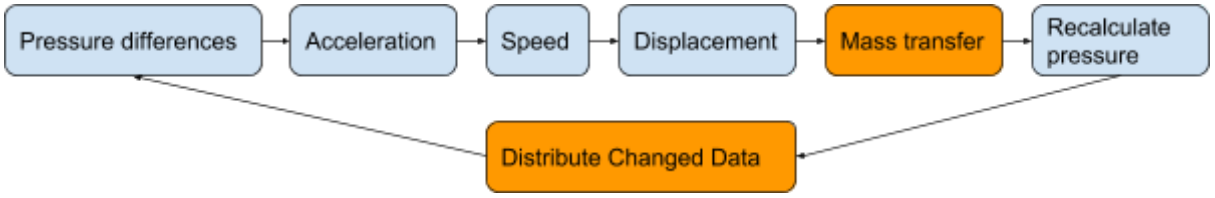

Figure 2: Communication bottlenecks (orange) in the air motion algorithm

Table 3: Algorithmic breakdown in relation to Foster's agglomeration criteria

Criteria	**Criteria Met?**
Agglomeration should reduce communication costs by increasing locality	Yes
Any computation replication should not outweigh its costs	Yes
Any data replication should not limit scalability	~Yes
Should produce tasks with similar computation and communication costs	Yes
The number of tasks can still scale with problem size	Yes
There is sufficient concurrency for target/future computers	Yes
The number of tasks cannot be made smaller without introducing load imbalancing/increasing development costs/reducing scalability	Yes
The trade-off between the chosen agglomeration and cost of modification to existing code is reasonable	N/A

Table 3 shows to what extent Foster's checklist for the agglomeration element of the design process was met.
Communication costs have been reduced by opting to arrange processes in a 3-dimensional taurus, and having processes only communicate with their neighbours.
There has been no computation replication, thereby fulfilling the requirement not to increase costs through replication.
In any reasonable simulation, the data replication would have very little impact on the ability of the algorithm to scale. However, in extreme cases of the CUDA implementation, the requirements of the data to be copied to the device would result in an upper limit. The MPI

implementation, if run in a distributed manner, fully meets the criteria of data replication not limiting scalability.
In both instances of the algorithm, all tasks perform near-identical computations and communications.
Concurrency may always be added to the system by reducing the volume of each subvolume, thereby increasing the number of subvolumes being simulated. Equally, the size of the problem can be increased trivially by increasing the size of the atmosphere being simulated. In doing so, the number of tasks available increases and therefore the implementation allows for sufficient concurrency for both the target computer (in this case a desktop computer with a GTX 680 graphics card), and almost any future computer. The algorithm would suffer in increased complexity if the tasks were further reduced in size, due to the difficulties in synchronizing access to each subvolume to perform each calculation being performed.

Mapping

Both the CUDA and MPI algorithms are constructed from the same basic premise; each thread (CUDA) or process (MPI) deals with a single subvolume in parallel, synchronising the communication between threads or processes as required to ensure the produced data is accurate. The main difference between MPI and CUDA is the method by which the information is communicated and this has a direct impact on the way in which the tasks are assigned to processors. Communication is required for the transfer of mass between subvolumes and the synchronization of mass changes at the end of every timestep, fig 2. In the MPI implementation processes are arranged in a 3D taurus. Mass transfers are achieved through direct communication between neighbouring processes. For the synchronization of the entire atmosphere at the end of each timestep, messages are sent to the process with rank 0. From process 0, the data redistributed to all processes. A mesh network was chosen due to the fact that the cost of scaling remaining constant, and being relatively simple to implement (Gropp, n.d.).
In CUDA, planes of subvolumes are separated across a one dimensional grid of blocks, in which a two dimensional array of threads each process a single subvolume. In order to synchronize the atmosphere, each thread accesses a different element in a global array to store the updated information for the subvolume it is concerned with, at the end of each timestep. For air mass transfer, a thread may directly accesses a neighbour for communication, with syncronised access to memory to ensure no race conditions may occur. Given an even distribution of air movement (i.e an equal chance air is transferred up, down, left, right, forwards or backwards from a subvolume in a timestep), there is a two in three chance of a local, non-cross-block communication.
Also, due to the decomposition performed, the number of communications performed by each element remains fixed irrelevant of the project size. This is because air may only travel in one direction and only to adjacent neighbours in each timestep. Therefore, in the worst case, there is a maximum of N communications performed and a best case of 0 when the system is perfectly balanced (however this is exceedingly unlikely), where *N* is the number of subvolumes in the dynamical system, excluding the synchronization at the end of each timestep.

Evaluation

Two experiments were conducted on each algorithm to evaluate their performance. In both tests, clock ticks were measured for the duration of the simulation and then converted to a value in seconds using the number of clock ticks per second. The resultant timing value was measured to three decimal places. For CUDA, the timing was performed from the invocation of the kernel until the result was copied from the device back to the host. In the case of MPI, the time was taken from the call to the 'Simulate' function until the process with rank 0 returned from the function, which included the final collation of data back to rank 0.

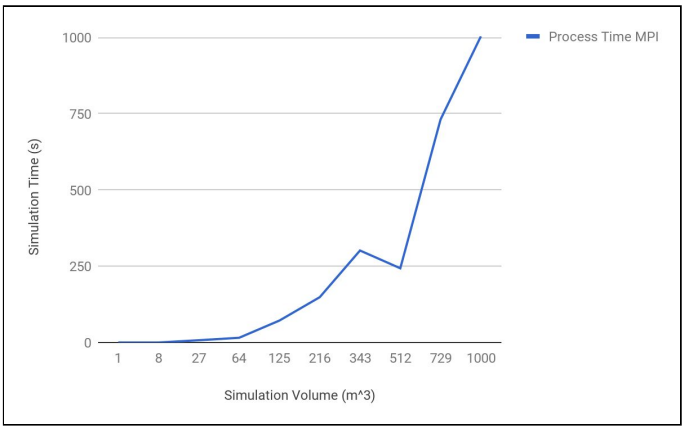

Figure 3: Simulation volume effect on processing time in MPI

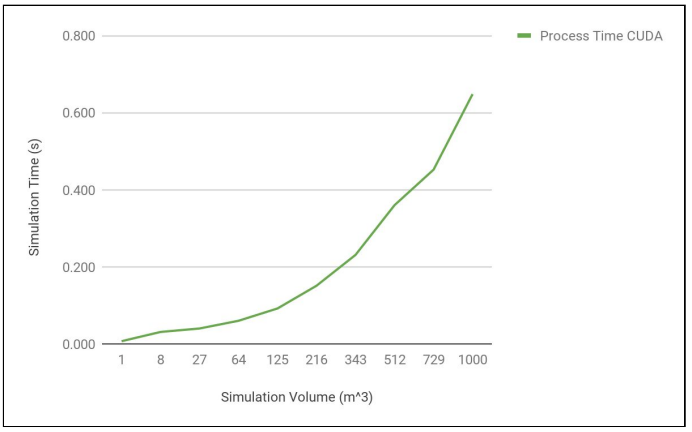

Figure 4: Simulation volume effect on processing time in CUDA

Firstly, scalability was tested by varying the size of the simulated atmosphere whilst the number of iterations remained fixed. As can be seen when comparing fig. 3 and fig. 4, both CUDA and MPI iteration times increased with a similar profile, with the time consistently increasing as the size of simulation increased, with the exception of the 512 m^3 simulation in the MPI experiment. The present study postulates that due to 512 being a 2^n value, some internal MPI optimisation was taken advantage of and therefore reduced communication overheads in that instance. Although both algorithms had a similar profile, the MPI implementation saw an increase in six orders of magnitude in the time taken to process the simulation between the first and last test. In comparison, the CUDA implementation performed better in all instances of the tests except for the trivial instance of $1m^3$. The CUDA algorithm also saw a far lesser increase between the smallest and largest simulation with an increase of only two orders of magnitude.

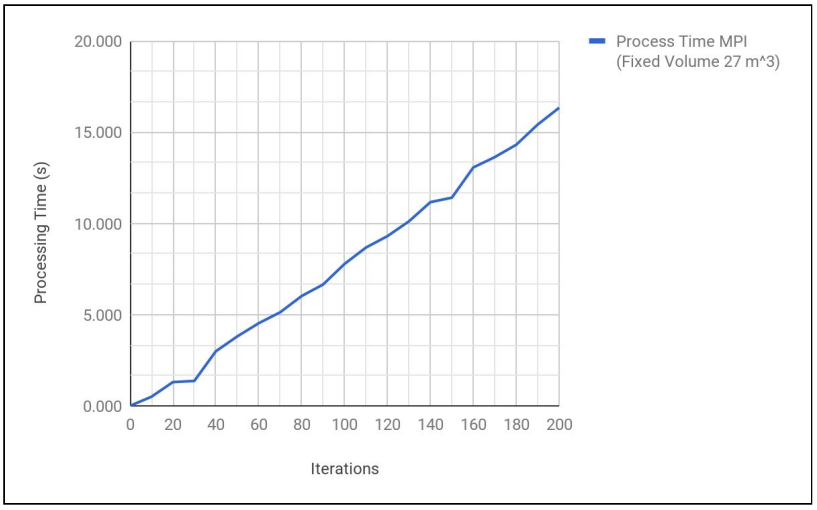

Figure 5: Simulation iteration count effect on processing time in MPI

Secondly, process time was measured whilst the number of iterations was varied. The size of the volume was fixed at $3m^3$ to keep testing times reasonable, based on the results of the first experiment. Initially a run of zero iterations was conducted to measure the setup overhead for both CUDA and MPI. This resulted in an average overhead of 0.017s and 0.000s respectively. The lesser overhead cost for MPI is to be expected, as there is an inherent overhead in launching a kernel that is not present in the equivalent function call in MPI. However the communication in the MPI implementation introduces overheads that can be clearly seen in both experiments in the increased times. It is also possible that the MPI implementation was limited by overheads introduced through running all MPI process on a single node. This caused significant memory consumption (upwards of 90%) in several tests, and likely negatively impacted the performance of the algorithm. As such, running the MPI algorithm in a distributed manner across several nodes may see a significant increase in performance, if not offset by increased communication time overheads.

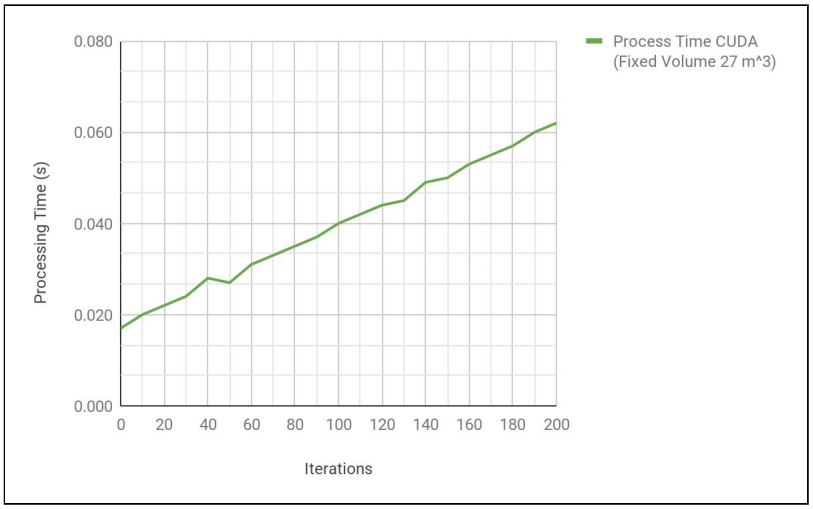

Figure 6: Simulation iteration count effect on processing time in CUDA

Both MPI and CUDA algorithms showed a linear increase (fig. 5 and fig 6) in processing time with respect to the number of iterations required. However, as with the previous experiment, cuda processing times were orders of magnitudes faster in non-trivial instances, with the slowest average processing time of 0.062s in comparison to MPI's 16.339s, a difference of ~263.5 times.

Conclusion

Overall, both algorithms can successfully process and simulate the dynamical system at a large scale. However, the CUDA algorithm shows a significant advantage over the MPI equivalent in both test scenarios. This is likely due to the lack of warp divergence in the CUDA implementation. Nonetheless, improvements could still be made to the CUDA algorithm. For example, warps may be underutilised in the current implementation due to the fact that block dimensions are calculated purely from the dimensions of the atmosphere being simulated.

Similarly, improvements could be made to the MPI algorithm. More communication is used than is required in the present implementation of the MPI algorithm. This is due to limitations in the MPI framework require that create a situation in which a mass transfer of zero is transmitted to neighbours when no transfer is required. This is because there is currently no way for a process to calculate which neighbours need to transfer mass into the subvolume it is managing. Even without the outline changes, it may be that MPI can perform better than has been shown in the present study. This could be achieved by running the algorithm over

multiple nodes, thereby significantly decreasing the memory limitations seen during these experiments. Further, an implementation in which the CUDA algorithm is embedded in an MPI distributed system may show additional increases in performance for larger problem sizes by benefitting from the strengths of both algorithms; distributing large subsections of the atmosphere to separate nodes via MPI, on which CUDA is utilised to do the heavy lifting before passing the subset back to MPI to collate the data for the next timestep.
Finally, it is worth noting the differences in scalability costs as each algorithm reaches its limits. In the case of the MPI algorithm run on a single node, where this study has found memory to be a significant limiting factor, the cost of scaling hardware is relatively low, due to the addition of more RAM increasing the potential problem size the algorithm is capable of. Conversely, in CUDA, as the algorithm reaches its limits, a better graphics card would be required, requiring upwards of 10 times the cost for improvement.

Future Work

The present study would be well supported by further work investigating the combination of the CUDA and MPI algorithms. It may also be beneficial to compare the CUDA, MPI and mixed implementations with a serialized implementation to better compare costs and overheads for all parallel implementations. Further, the analysis of implementation time and detailed memory profiling of the implementations was out of scope of the current study and extended investigations in these areas may better inform the decision of which framework is best suited to air motion dynamical systems.

References

Atkinson, B. (1981). *Dynamical meteorology*. London: Methuen, pp.4-5,12,21.

Gropp, W. (n.d.). *Process Topology and MPI*.

Hanrahan, V. and Porkess, R. (2014). *Additional mathematics for OCR*. London: Hodder Education.

Holton, J. (1992). *An introduction to dynamic meteorology. 3rd ed*. San Diego: Academic Press, pp.7,475.

Zdunkowski, W. and Bott, A. (2004). *Thermodynamics of the atmosphere*. New York: Cambridge University Press, p.6.