Oliver Goldstein: og14775
Please ensure this is printed in colour

Lattice Boltzmann Code: MPI Optimisation

Introduction

The first step was implementing a halo exchange. I made an educated guess as to where to start. Initially, I split the grid row wise (as C is a row major language) with each core receiving northern and southern halo rows (with the southern and northern directional components respectively). Each core receives no less and no more than the amount of memory it needs for both the obstacles and the grid itself, this is especially relevant as on the largest grid, each array within the struct of array implementation I have, can be larger than each processors L1 and L2 cache sizes combined. This decision was motivated by the desire to optimise for spatial locality in the SOA (Struct Of Array). The halos are exchanged in four steps. The exceptional halo rows are exchanged in a peeled loop to optimise for vectorisation, in the sense that fewer branches occur and unit stride exists. The exchange mechanism is:

1. **Odd** ranks **receive** southern halos whilst **even** ranks **send** their southern values.
2. **Even** ranks **receive** southern halos whilst **odd** ranks **send** their southern values.
3. **Odd** ranks **receive** northern halos whilst **even** ranks **send** their northern values.
4. **Even** ranks **receive** northern halos whilst **odd** ranks **send** their northern values.

Send and receive operations were synchronised at this point, using MPI_Ssend. The initial scalings for different grids and core counts with mpich are plotted in Fig 1.3. In the following section, I will construct a list of hypotheses that, given time, I will attempt to test.

Fig 1.0 Communication: Tiles vs Rectangles

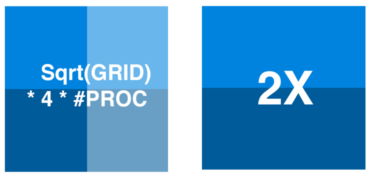
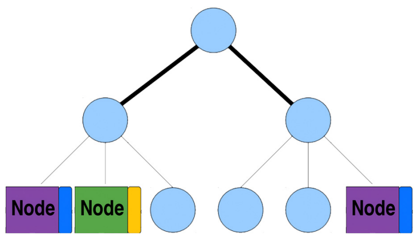


Fig 1.1 Node connection diagram of BlueCrystal P3

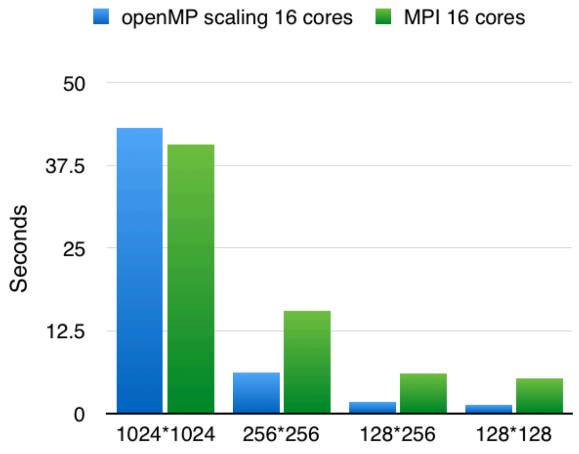


Analytical A Priori Hypothesis Construction

In Fig 1.3, one can see that as core count increases, for the smaller grids most especially, relative speed up decreases. I contest that this is due to the communication overhead, especially if the MPI implementation fails to place neighbouring virtual grid spaces, physically next to each other on the physical cores within BlueCrystal. Running an experiment with just computation or communication on 64 cores, with a grid size 128 * 128, communication represents 98% of the time. This relationship can be seen in Fig 1.9. In addition; as we approach a greater number of nodes, grid tiles are physically further away from each other. Given the speed of light is only ~three hundred million metres per second and nodes can be up to ten metres away from each other in the case of BlueCrystal, often the sheer distance and the relative processor clock speed leads to many tens of wasted clock cycles.

In the worst case, every neighbouring virtual processing region could be on a different node, with each processing region in Lattice Boltzmann having 2 neighbours. If the root of the FAT tree network within BlueCrystal has a high degree of congestion, messages may have even more latency. Although the job scheduler attempts to put them close to each other, there is no guarantee, so in the tradition of algorithmic analysis one can examine a worst case situation. Packets of data would traverse up and down the tree in 2log(n) time such that in BCP3, with approx 400 nodes, this may take 10-20 routing decisions (base dependent). In addition if they are not neighbours within individual sockets/dies there could be an added delay. If this occurs at each time-step this could be very costly. A visual representation of Blue Crystal's structure can be seen in Fig 1.1.

Fig 1.2 OpenMP vs MPI (Av. 2 runs)



Potential Benefits of A Tiled Cartesian Topology

Tiling the virtual grid has two benefits. The first is that the difference between the quantity of halos needed and the internal processing region of squared size tiles is far greater, in the sense that, per cell, far less communication is needed at each time-step. Fig 1.0 depicts that as N, the number of processors grows, the communication decreases asymptotically faster in a square arrangement. This scaling can be seen in Fig 1.4, showing that asymptotically, tiling performs far better in terms of communication overhead. The Y axis depicts the amount of communication that must occur and the x axis shows the size of the (square) grid. The row wise scaling has such a steep gradient that it is barely visible against the y axis. Row wise for 1024 at 64 cores is roughly 131,000 whereas with tiles it is only roughly 9,000. If additional cores are added this relationship is exacerbated.

In addition to the above, using an MPI 2.0 cartesian topology, MPI can place the virtual tiles such that, in the physical hardware, virtual tiles are next to each other, minimising communication distance.

Fig 1.9 Computation vs Communication on 128 * 128

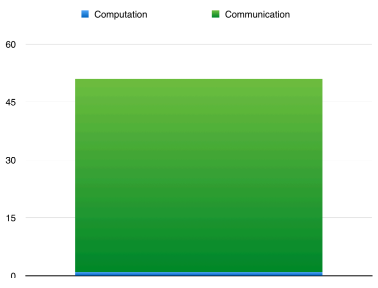


Fig 1.3 Initial Scaling on different grids. (Av. 2 runs)

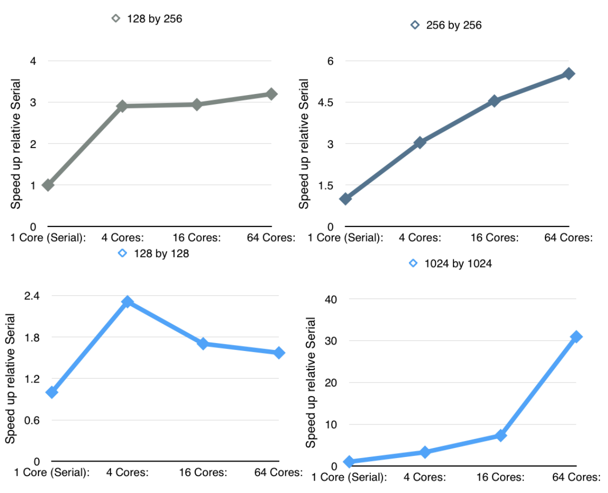


Fig 1.4 Scaling 64 Cores

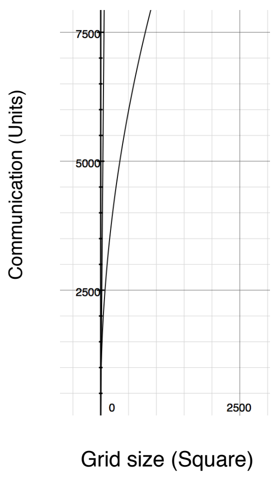


Fig 1.9 Tile wise split (Constant Slowdown)

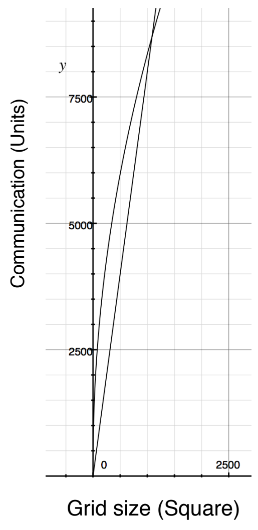


Fig 1.5 SMP Placement

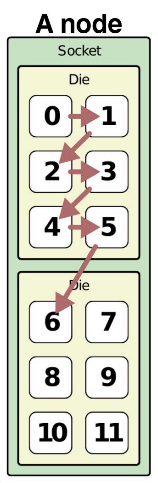
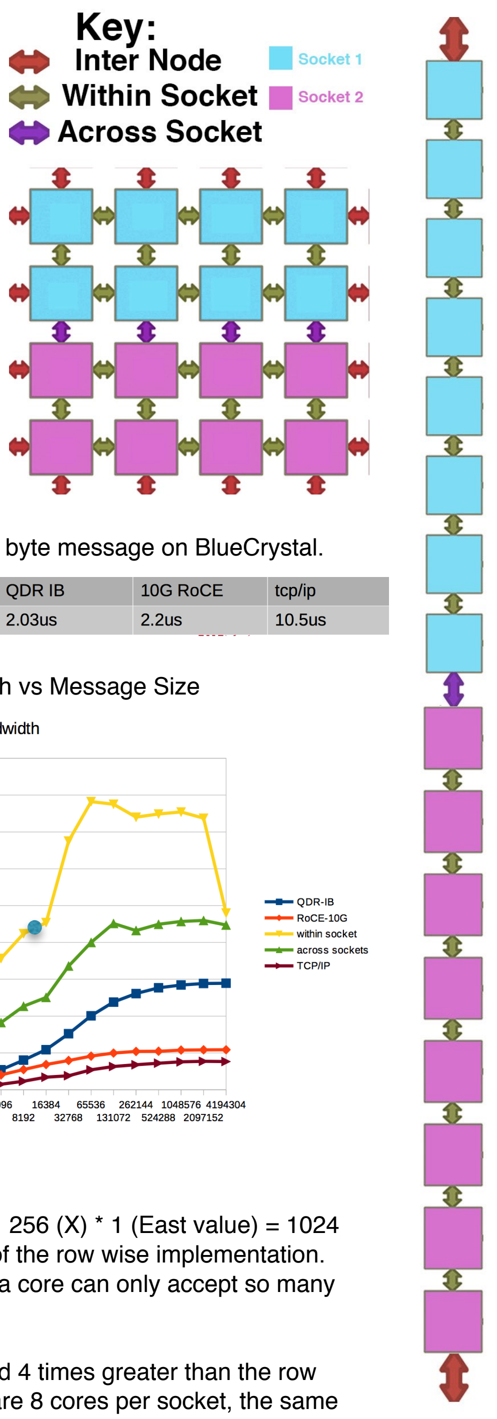


Fig 1.6 Grid Analysis, Tiled vs Row Wise



By default, MPI places ranks in a sequential manner i.e. SMP-style placement (environment variable MPICH_RANK_REORDER = 1), such that consecutive ranks are placed on consecutive cores. Fig 1.5 depicts this relationship. However, in the initial row wise implementation, consecutive ranks are neighbours in the grid. This may negate the relative benefit, I experience, of a cartesian topology.

Fig 1.7/1.8/1.1:
Dr Gethin Williams @ UOBristol
Simon Mcintosh Smith @ UOBristol

Fig 1.7 Message Latency for 0 byte message on BlueCrystal.

	within-socket	across-sockets	QDR IB	10G RoCE	tcp/ip
0 byte msg	0.12us	0.25us	2.03us	2.2us	10.5us

Fig 1.8 Bandwidth vs Message Size

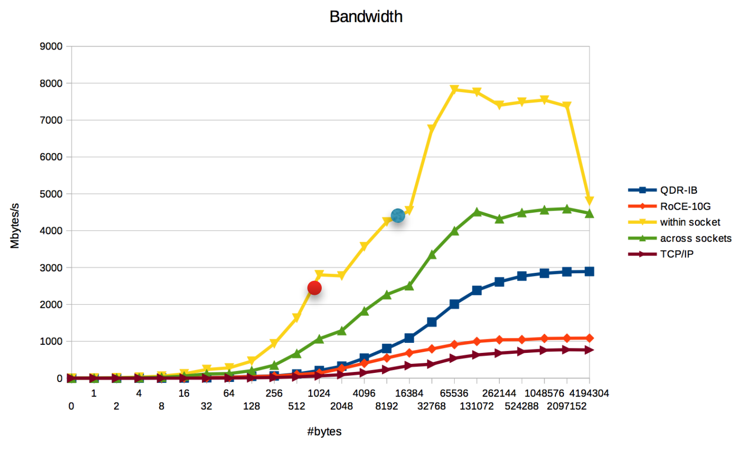


Image: Dr Gethin Williams @ UOB

Potential disadvantages of tiling

Smaller messages would be sent in the tiled system that I would implement, given the time I have, relative to the row wise implementation. This means that multiple headers are sent representing a greater overhead. For instance; on the 1024*1024 grid with 64 cores, assuming packed messages, the size of each message, in the smallest case for the row wise layout would be: 4 (Float size) * 1024 (X) * 3 (Northern values) = 12,288 bytes, corresponding to the blue point in Fig 1.8 (Within Socket). For the tiled layout, the smallest message would be: 4 (Float size) * 256 (X) * 1 (East value) = 1024 bytes, corresponding to the red point in Fig 1.8. The memory bandwidth is approximately half of the row wise implementation. This implies a smaller throughput and a greater latency of the system as a whole, especially if a core can only accept so many communications at once. This may be greater for internode communication.

In addition to the above, the number of inter node and across socket messages are 8 times and 4 times greater than the row wise implementation, respectively. To see this relationship; see Fig 1.6, which assumes there are 8 cores per socket, the same number as Blue Crystal. In Fig 1.7, for a zero byte message, one can see that across socket latency is double the within socket latency. The internode connection could conceivably be many orders of magnitude higher than this. I did not have time to empirically verify this. This is a massive limitation of a tiled approach.

Hypothesis #1: A combination of tiling and a cartesian topology will increase speed.

Shared Memory Versus Message Passing

For a grid of size 128*128, each time-step communicates, in total, almost as much as the allocated grid itself. In a shared memory parallel implementation such as openMP this coordination within a core is a highly optimised process managed by the hardware and potentially, the operating system through shared memory, coupled with a fast caching mechanism (assuming false sharing doesn't occur), library routines and directives. An empirical result of the initial speed difference can be seen in Fig 1.2. This shows **at first glance** that particular openMP implementations can do better than a naive implementation with mpich. However, openMP does not have the ability to transfer information over a network of nodes. This implies the slowdown I have experienced may be due to communication, which is expensive and minimising communication between nodes could be effective in improving performance. If further MPI optimisations yield no return, then I will use openMP on each node and MPI only to transfer data between nodes.

Hypothesis #2: Does a hybrid implementation with openMP improve performance.

Asynchronous Communication.

Potential limiting factors

I challenge the utility of asynchronous send and receive operations. Generally the main processing regions are a priori balanced, with each processor doing the same amount of work. Accelerate Flow represents an imbalance, but it is relatively small with respect to the main processing loop. In addition, the grid is a highly dependent system, with a high amount of communication needed, often. After the number of time steps equal to the number of processors in total, every process will have been affected by the propagation of northern or southern rows of every other process. This implies that at first glance, given a fixed size halo exchange, at most, the ability for DMA calls such as MPI_Put or asynchronous methods such as MPI_Irecv, are limited if a single processor lags behind the others. It may imply that a processor may only be able to progress n rounds ahead of the lagging process for all processes, such that n is the number of processors in the system. One might question the ability of such an imbalance to arise, but given hardware fluctuations or bad placement in the FAT tree, a significant relative slowdown may occur.

Potential advantages

On the other hand of the argument, by processing the internal grid region whilst the halos are being sent asynchronously, the latency will be effectively hidden, meaning as long as the communication time is less than the computation time of the internal grid region, communication is ‘free’. This can be seen in Fig 2.1 for a square grid region. If no one process, within the grid, systematically lags behind, then this scheme may be very effective.

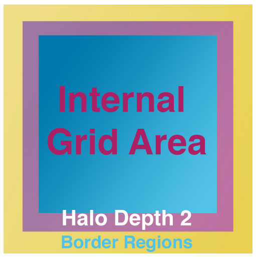
If the relative progress of each processor has a high variance, in the sense that processors change their relative progress with respect to each other very often, then I suspect asynchronous methods can be more effective. I deduce this, as the assumptions I made above, are based on the idea that the slowest processor remains the slowest processor throughout processing. If the fastest and slowest members change position frequently, then overall the system will move faster than if the system always waits for one particular slow processor.

In order to understand the potential improvement that could, in theory, be gained, I first ran the code such that communication was the only element that occurred in each iteration in order to ascertain the percentage of time that communication represented. This can be seen in Fig 2.2. At this point the time on 1024*1024 was almost 10 seconds, so roughly 5 seconds could be removed from the total time. The true ratio is hard to measure due to dependencies between communication and computation. The most important factor I felt, was ensuring communication time was less than the computation time. If this was not the case there may be no difference in speed. I will first avoid buffers and use Irecv/Isend rather than Bsend due to the well known latency associated with buffers.

Hypothesis #3: Do asynchronous calls improve performance.

Message Size

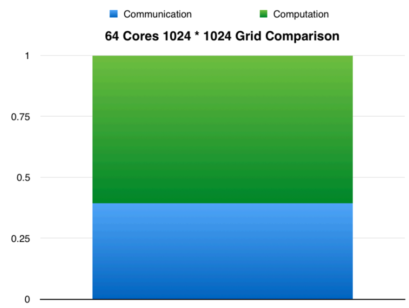
Fig 2.1 Structure of asynchronous communication.



Initially, rows of specific directional components of cells are transferred between ranks. I decided to implement a struct of arrays rather than an array of structs, such that memory regions are contiguous. This allows for efficient send and receives to occur without time spent packing messages with derived data types or unnecessary computation that would be achieved with an AOS. The sizes of the messages are also important as communication bandwidth increases as the size of the message increases (albeit non linearly). This occurs up to the point where message size exceeds the cache size, (see Fig 1.8). The increase in bandwidth may be due to fixed size headers which direct the message through BlueCrystal or communication setup costs, which represent a significant overhead for small messages. Upon sending messages, I will modify the data structure such that the northern and southern arrays values are contained in a single contiguous array and send that. I may have to pack messages into a buffer with memcpy. This is only effective if the communication time is longer than: **the computation** + ((**the time it sends to pack messages together**) / **overall speedup of increasing the message size**).

Hypothesis #4: Does increasing message size reduce communication time.

Fig 2.2 Communication vs Computation ratio



Message Structure

In order to further amortise the cost of sending data, I want to send two or more ‘fat’ halos at once. A fat halo is one whereby over one halo row is sent (see halo depth in Fig 2.1). Using this extra row, with a trade off of repeated computation, the next round over, I do not need to exchange halos, as they can be calculated locally on each core, implying half (or potentially more depending on the width of the halos transmitted) as much communication needed. The ideal point is met where the cost of the repeated computation per round is less than the sending of the halo row.

Hypothesis #5: Does the usage of halos of depth 2 halve communication time.

Informal Empirical Analyses of Hypotheses

Hypothesis #1: Implementing the tile wise split resulted in a speed up only on the largest grid, with the largest number of cores of only 15% (~10s to ~8.5s). On all other grids, the tile wise split took much longer to compute, for instance on the 256*256 grid, it took twice as long with 64 cores. This can be seen in **Fig 1.9** above. I suspect the constant factor slowdown due to increased inter-node communication and the smaller messages caused a slowdown within the system. However, testing on a handmade grid of size 2056 * 2056, yielded a time which was half as long relative to the row wise split. I therefore **reject** hypothesis #1, with respect to the “artificially” small problem we have been given. In conclusion, the internode communications and decreased message size only adds a constant factor. For large grid sizes with many processors (up to a limit with respect to Amdahl's law etc), tiling the grid will be asymptotically faster than a row wise approach.

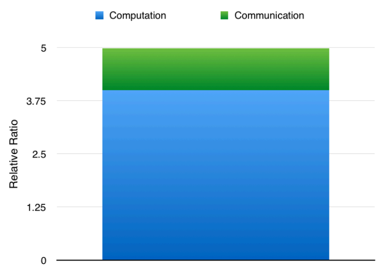
Hypothesis #2: I did not have enough time to implement a hybrid openMPI/MPI implementation. However, after I did alternatively experiment with send_recv and found that by allowing MPI to piggyback requests messaging was somewhat faster. I can conclude that upon further experimentation with MPI, the differences between the implementations have narrowed massively and so is not such an interesting experiment. I **reject** the hypothesis purely on the fact that after further optimising MPI, it outperformed openMP.

Hypothesis #3: Overlapping communication and computation resulted in an immediate reduction of almost a third to around 6.4 seconds. This is averaged over 3 runs to account for the nondeterminism of node placement and is a statistically significant difference. Latency is now less expensive when it occurs asynchronously. I suspect this process is highly dependent on the hardware involved as there is no guarantee of true asynchrony, it may only be non blocking. Initially I used the MPI_Waitall function before processing any of the rows. Given more time, I would have used the MPI_Waitany function, which waits for either the northern or southern halos to complete and works on the one transferred first, whilst still asynchronously waiting for the other. I therefore **accept** this hypothesis.

Hypothesis #4: In order to pack messages together, I edited the SOA data structure, such that instead of separate structs for the north western, northern and north eastern values (vice versa for south), I packed them into one contiguous array. For the northern most halos I had to send the top row only. Initially, I attempted a layout such that for the northern array for instance all values mod 3 where north eastern, mod 2 north western and multiples of 3 northern. This would imply I could simply send the array as it was without a handmade buffer. Unfortunately, due to programming related bugs, I was unable to implement this. Instead, I had a contiguous array with each of the values separated by offsets and using memcpy, I transferred them into a buffer. Fortunately, as memcpy was incredibly fast (perhaps due to an efficient implementation especially with respect to vectorisation), the time spent packing them into an array was worth it as the corresponding increase in bandwidth and therefore performance was great. I therefore **accept** this hypothesis.

Hypothesis #5: After implementing halos of depth 2, I found that the execution time did not decrease. In fact, the execution time increased by 5%. Running an experiment showed the communication computation ratio on the largest grid was now more or less optimal, see Fig 2.3. The only sensible conclusion I can make is that in order to compute halo rows, one requires a lot of data from the above two rows, including the obstacles grid. This required 9 different memcpy operations in total, a typecast from obstacle unsigned chars to floats on each communication end and 3x the amount of data. This is far more than the 2 memcpy operations for halos exchanged each time and even in light of potentially improved memory bandwidth, did not compensate. I therefore **reject** this hypothesis.

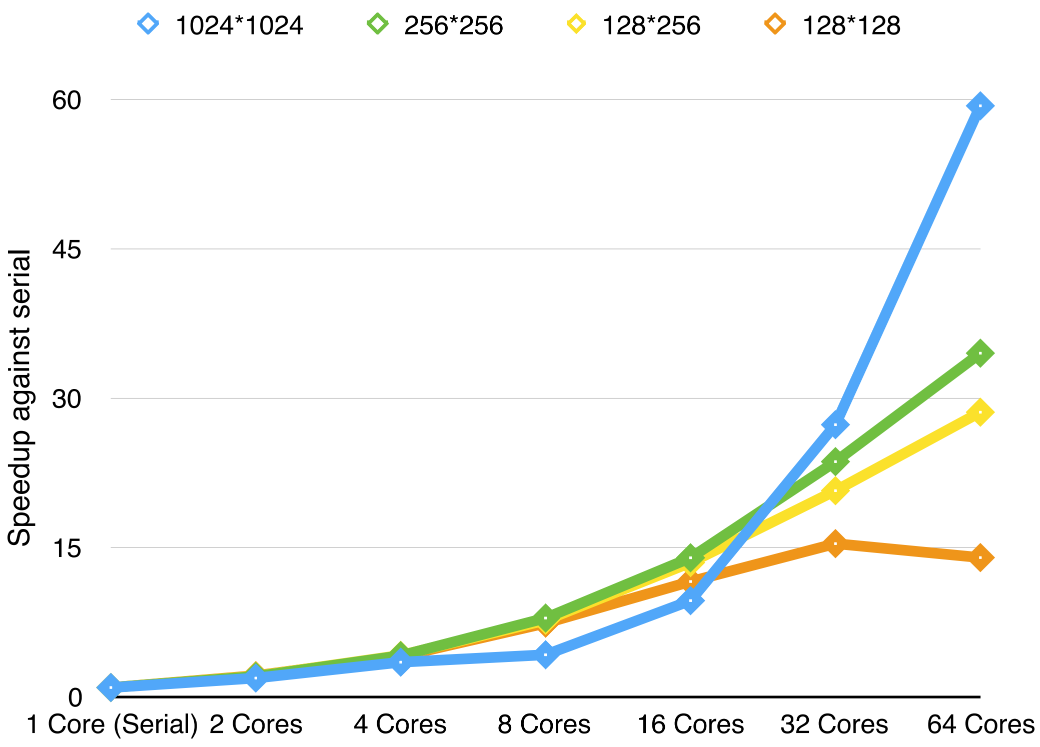
Fig 2.3 Computation vs Communication ratio on 1024 grid with 64 cores.



<u>Conclusion & Final Results</u>

The smaller grids do not scale as well as with many cores there is relatively far more communication. All of the grid sizes scale well for core counts up to 4. The largest grid dips at 8 cores, perhaps due to random fluctuations, although I am unsure as to why this is. The largest grid scales the best. After testing on a grid which was four times the size of the 1024 * 1024 grid there was an improved scaling still with a 63.5 times speedup. The most significant optimisation was the overlapping of computation and communication. Overall all grid sizes improved their performance and scaling. With the 128 * 128 grid, one can see a real life occurrence of Gunther's law, which states that speedup is not linear with core count and eventually tails off due to queueing contention, coherency delay and communication.

Fig 2.4 Final speedup for all grid sizes on power of 2 core count. (Av 2 runs) relative to optimised serial code.



<u>Evaluation</u>

Given more time, in future, I would have increased the depth of halos transmitted. I would have tiled the grid with an array of struct of arrays to improve the simplicity of tiling. I would also have experimented with different flavours of MPI communication schemes. N.B. The code I provide does not work with an odd number of cores, in order to do so it would have been a trivial redistribution with the mod operator. I would have attempted to make them as load balanced as possible. In addition, I would have used tools like VAMPIR, Perfex or simple MPI_Stats in order to analytically evaluate the performance of the MPI calls being made. I could have used an array of struct of arrays in order to tile the grid more efficiently, which may have affected the result of hypothesis #0.

Fig 2.5 Final results for all grid sizes on 64 cores. (Av 2 runs)

Grid Size:	Final Time (s)	Final Speedup (Relative to optimised serial)
128 * 128	0.66	14.0
128 * 256	0.63	28.6
256 * 256	2.05	34.5
1024 * 1024	5.15	59.3