José Hernández del Pino (jh3875@bristol.ac.uk)

Distributed Lattice Boltzman using MPI

Introduction

In this document we will present the various techniques used to parallelise and distribute across a cluster an already optimised reference serial C implementation of the lattice Boltzmann scheme. Our optimised serial code is easily parallelisable because the data dependencies have already been removed as far as the implementation would allow. That is to say:

- Source code has been restructured to maximise the number of potentially parallelisable regions (Chapman, Jost, & van der Pas, 2007).
- Cells can read but do not modify neighbouring cell values to avoid unnecessary synchronisation (Bella, Filippone, Rossi, & Ubertini, 2002).
- Input and output buffers are switched intermittently between time steps using just pointers to minimise memory copying.

Table 1 presents the output obtained by running the optimised serial version of our code under the TAU profiler. The table shows that the program spends most of its time in the *collision, average velocity and flow acceleration* functions.

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	47	1:27.994	1	20004	87994913	main
76.9	51	1:07.651	10000	20000	6765	timestep
76.5	1:07.307	1:07.307	10000	0	6731	collision
22.9	20,131	20,131	10001	0	2013	av_velocity
0.3	293	293	10000	0	29	accelerate_flow
0.2	158	158	1	0	158041	write_values
0.0	5	5	1	0	5790	initialise
0.0	0.003	1	1	1	1489	calc_reynolds
0.0	0.569	0.569	1	0	569	finalise

Table 1. Profile output from the optimised serial code.

Data distribution

To allow our program to run on a distributed environment we need to devise a mechanism to split the data into tiles across instances of the program running on the cluster so that each instance performs calculations on just a subset of the data. In addition to this, a strategy to exchange shared data across multiple nodes is also necessary. There are a number of possible layouts to segment the data in different ways for distribution across the network. In this exercise we have considered three, namely a grid (squared), a vertical and horizontal splits.

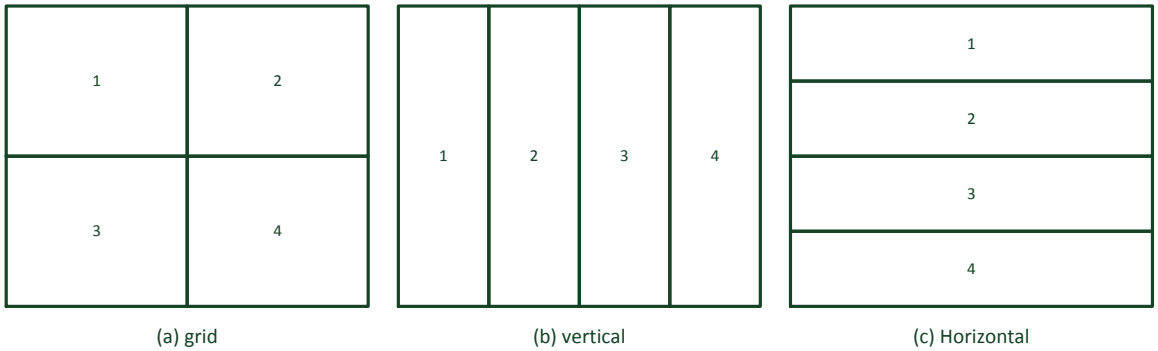


Figure 1. Possible data layouts to segment the data for distribution across the network.

The guiding objective in the selection of the best split for the data is to minimise the sources of overhead, that is, communication, idle time and extra computation (Pacheco, 1997) that may be incurred by each of these segmentation approaches. Based on communication, we eliminate the grid (squared) layout because this split requires us to communicate with all neighbouring process instances in order to exchange the shared data shown in Figure 2, resulting in $8 \times n \times m$ send/receive message pairs, where n is the number of rows in the grid and m represents the number of columns.

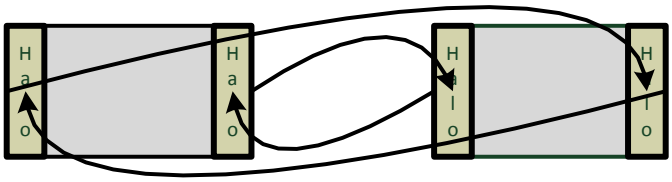


Figure 2. Wrap around halo (shared data) exchange between two neighbouring cells using two cores.

By contrast, if we were to split the data into columns or rows as shown in Figure 1 (b) and (c) we only require $2 \times m$ or $2 \times n$ message pairs respectively since each process now only has two other neighbouring processes to communicate with. The horizontal arrangement offers one further advantage over its vertical counterpart in that the halo or exchange buffers are already aligned into rows of contiguous data, making the column to row transformation unnecessary and thus minimising the amount of extra computation necessary prior to the transmission or after the reception of data. For these reasons, we have chosen to use the horizontal data split.

For small scale problems such as ours (300x200) where the communication time may exceed the computation time, the horizontal arrangement seems more appropriate. In larger problems however this data split can limit the capacity to scale-up because it is bound to the number of rows and cannot therefore scale well when the number of columns alone is increased. The grid (or square) split does not suffer from this limitation since its dimensions can vary independently and yet allow for more processes to be added to the cohort.

Outline of the distributed algorithm

The general outline of our Boltzmann MPI algorithm is as follows:

1. Initialize MPI
2. Read input parameters and obstacles from file in master process and broadcast to all other processes
3. Initialize static send/receive buffers and permanent communication channels
4. For each time step
    a. Accelerate flow
    b. Simultaneously initiate halo exchange and perform calculation on non-shared data
    c. Wait for calculation and halo exchange to complete
    d. Perform calculation on the shared data
    e. Store local velocity value
5. MPI reduce the velocities from all processes to process 0.
6. MPI gather final state from all processes to process 0.
7. Write results from process 0 to file.
8. MPI finalize

Strategies for improving parallel performance

To maximise processor usage, we perform as many calculations as possible locally in each process and avoid sending large amounts of messages between processes. The end result is that the master process only needs to recombine partial results. We also pre-compute the offsets or indices that have the same set of values from one time step to the next to avoid the computational expense of calculating the modulus and branching through 'if' or ternary conditional expressions.

Idle time between time steps is minimised by distributing the workload evenly across processes. Where even distribution is not possible because the number of processes in the cohort does not divide into the number of rows, we ensure that at most only one of the processes in underutilised and the rest of cohort share a similarly balanced workload.

To hide the effect of network latency we divide the collision calculation into two distinct steps. The first initiates the halo exchange asynchronously in a background thread (with MPI_Startall) and immediately begins the collision calculation involving the local non-shared data or inner ring of the tile whilst the halo is being exchanged. The second step simply waits for the halo exchange to finish (with MPI_Waitall) and then performs the part of the collision calculation involving shared data, i.e. the outer ring of the tile. The end result is a complete overlap between computation and communication that fully hides the cost of network communication.

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	32	7,308	1	20026.2	7308334	main
58.3	90	4,257	10000	50000	426	timestep
42.6	3,114	3,114	20000	0	156	collision
15.8	1,151	1,151	1	0	1151834	MPI_Init
15.7	4	1,148	1	2	1148444	initialise
15.6	1,143	1,143	2	0	571750	MPI_Bcast
12.4	907	907	10000	0	91	MPI_Waitall
6.1	448	448	10000	0	45	av_velocity
2.9	215	215	1	0	215174	MPI_Finalize
1.1	82	82	10000	0	8	MPI_Startall
0.8	61	61	10000	0	6	accelerate_flow
0.6	42	42	1	0	42071	MPI_Gather
0.1	9	9	0.0625	0	157056	write_values
0.0	1	1	1	0	1748	MPI_Reduce
0.0	0.525	0.525	1	0	525	finalise
0.0	0.138	0.138	5.1875	0	27	debug
0.0	0.0245	0.105	2	8	53	do_halo_init
0.0	0.0358	0.0494	4	4	12	send_halo_init
0.0	0.015	0.0315	4	4	8	receive_halo_init
0.0	0.0238	0.0238	8	0	3	MPI_Request_free
0.0	0.0165	0.0165	4	0	4	MPI_Recv_init
0.0	0.0136	0.0136	4	0	3	MPI_Send_init
0.0	0.0135	0.0135	1	0	14	MPI_Get_version
0.0	0.002	0.002	1	0	2	MPI_Get_processor_name
0.0	0.00156	0.00156	1	0	2	MPI_Comm_rank
0.0	0.00144	0.00144	1	0	1	MPI_Comm_size
0.0	0.00112	0.00112	1	0	1	calc_reynolds

Table 2. Profile output from the parallel MPI code using a 16 cores.

Results

The results shown in Table 3 were obtained by running the optimised serial version and the parallel MPI version of the software on BlueCrystal Phase 1 using one to 20 cores. The results of running the optimised serial version of the program have been included so as to provide a basis for comparison and allow us to calculate the speedup factor.

Please note that the results have only been averaged over 5 runs to rather than the conventional 30 so as to limit the impact on other BlueCrystal users. Consequently, without a larger pool of samples, the measurements shown here may not lay at the centre of the normal distribution as might be expected under the Centre Limit Theorem.

Excluding errors introduced by sampling over just 5 runs, Table 2 shows the MPI version of the program run on a single core appears to have exhibited similar if not better performance than the purely serial version. Furthermore, profiler output (not shown here) suggests the cost of the MPI calls to be almost negligible, leading us to surmise that the MPI libraries have been optimised for this particular scenario thus incurring just the cost of a function call. Other discrepancies are likely due to code rearrangements resulting in improved cached utilisation. E.g. the blocking\tiling mechanism used for the halo exchange.

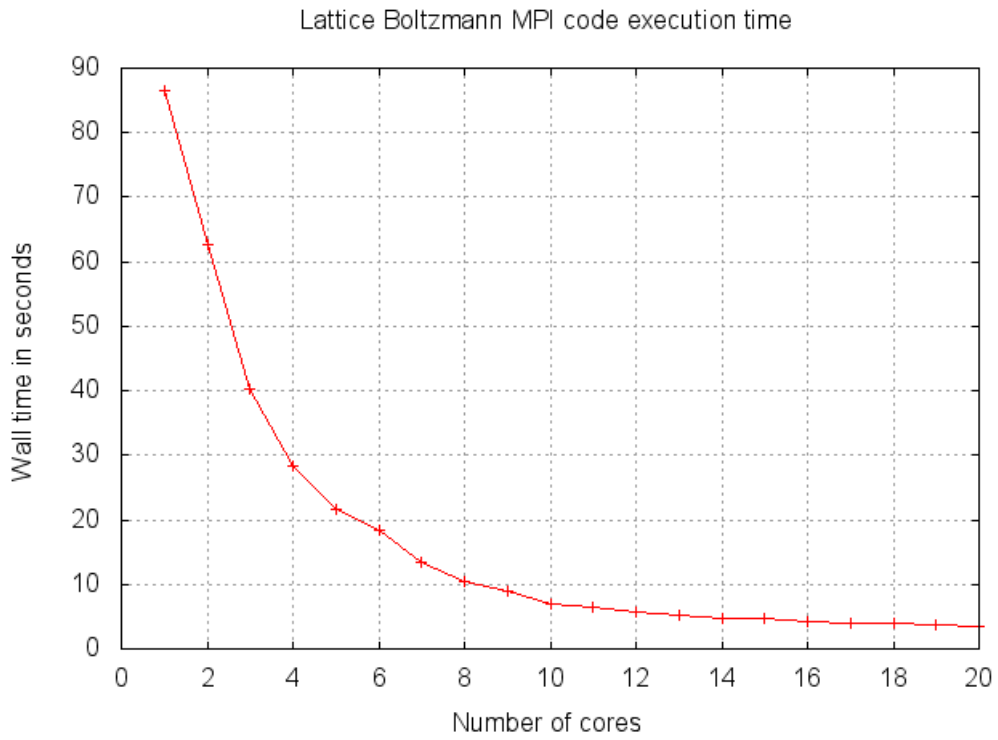
Program type	Number of cores	Wall time (in seconds)	Speedup
Serial (Optimised)	1	86.26	1.00
Parallel (MPI)	1	86.44	1.00
	2	62.65	1.38
	3	40.39	2.14
	4	28.25	3.05
	5	21.51	4.01
	6	18.34	4.70
	7	13.55	6.37
	8	10.44	8.26
	9	9.03	9.56
	10	6.87	12.55
	11	6.37	13.55
	12	5.69	15.17
	13	5.21	16.56
	14	4.84	17.81
	15	4.75	18.17
	16	4.27	20.21
	17	3.98	21.68
	18	3.96	21.77
	19	3.63	23.75
	20	3.40	25.36

Table 3. Execution time and speedup results

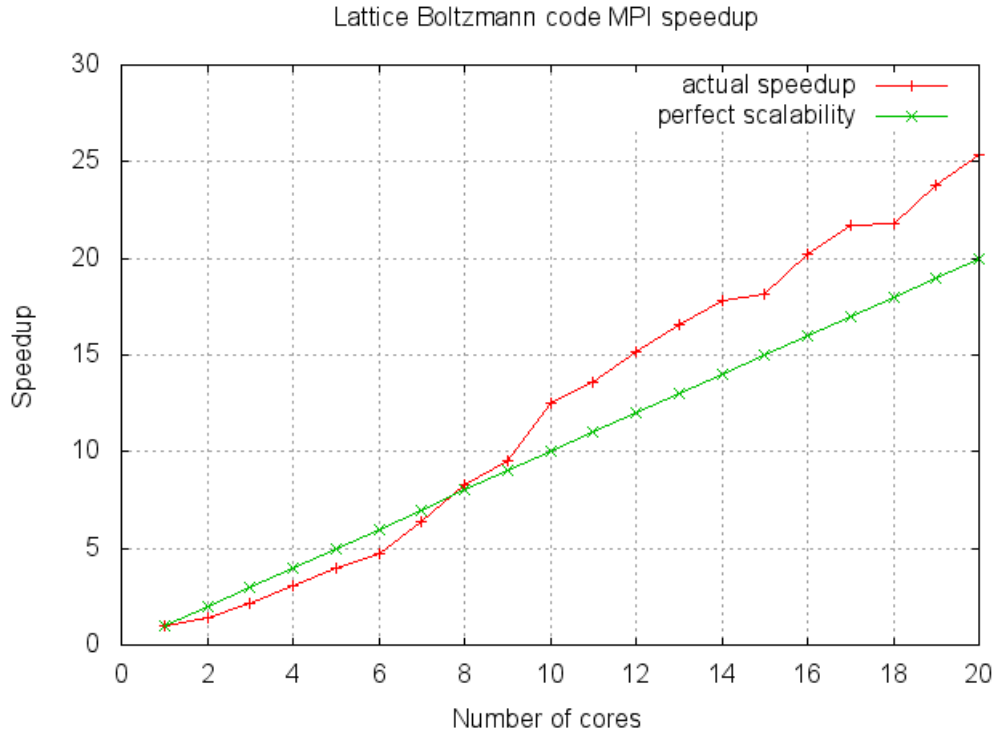
Our results showed a maximum discrepancy of 5^{-10} when compared to the reference results obtained by running the original unmodified program. Taking into consideration that the Reynolds number giving the relative balance between forces mimicked by the simulation remains unchanged, we deem this to be an acceptable error.

Jitters in the results can be due to various reasons. E.g. some processes may be underutilised depending on the partition size, some cores will have uneven loads when the node is occupied by processes from different users or even the physical location of the processor core in the cluster, or the sample population is not big enough.

Execution speed is initially reduced dramatically as new cores are added to the cohort, but this effect begins to tail off significantly after the cohort reaches a size of 10 cores from which speed increases are more modest. This is due to the fact that the workload reduction (or reduction in tile size) per core is now much smaller and the cost of managing both the division of work and added communication increases overheads.



Our implementation shows better than linear speedup when using more than 10 cores. This is probably due to improved cached utilisation on smaller tile sizes which is not taken into account by the linear speedup equation.



Bibliography

Bella, G., Filippone, S., Rossi, N., & Ubertini, S. (2002). Using openMP on a hydrodynamic lattice-Boltzmann code. *Proceedings of the Fourth European Workshop on OpenMP*. Roma.

Chapman, B., Jost, G., & van der Pas, R. (2007). *Using OpenMP Portable Shared Memory Parallel Programming*. Massachusetts: The MIT Press.

MPI Forum. (n.d.). *MPI Documents*. Retrieved 3 1, 2012, from Message Passing Interface Forum: http://www.mpi-forum.org/docs/mpi21-report.pdf

Pacheco, P. S. (1997). *Parallel Programming with MPI*. San Francisco, California: Morgan Kaufmann Publishers Inc.