Proceedings of the 24th UK Conference of the
Association for Computational Mechanics in Engineering
31 March – 01 April 2016, Cardiff University, Cardiff

Parallel implementation of numerical modelling of concentration polarisation and cake formation in membrane filtration processes

Bun Lo[1]*, Oubay Hassan[1] and Jason Jones[1]

[1]College of Engineering, Swansea University, Bay Campus, Swansea SA1 8EN
*839372@swansea.ac.uk

ABSTRACT

An existing model[1] capable of predicting concentration polarisation and cake formation in crossflow membrane filtration is implemented on different parallel platforms. The governing equations, Navier Stokes equations for the hydrodynamics and convection diffusion equation for suspended particles, are solved numerically using the lattice Boltzmann method. The equations are coupled through velocity, viscosity and diffusion coefficient, which are assumed to vary with the solute concentration.

The sequential algorithm is analysed to identify potential parallelisation sections and is implemented in C++ with OpenMP for multicore CPU, MPI for distributed memory CPUs clusters and CUDA for GPUs clusters. For each parallel implementation, optimisation is carried out too remove all identified bottlenecks. In general, these optimisations aim to fully utilise pipelines concurrently and eliminate unnecessary workload.

The parallelised algorithms are validated by comparing the results obtained for a number of problems with the predictions produced by other computational models and experimental results available in literature. With the current setup, the algorithms are both memory bound and compute bound, which allow them to scale well with increasing number of computing units. The paper will demonstrate the bottlenecks, effectiveness of optimizations and the performance of the parallel codes.

The parallel implementation of the modelling process demonstrates the potential of the computational technique to be deployed for the prediction of concentration polarisation and cake formation in a fast and efficient manner. Simulations can be done with higher resolution in a much faster timeframe, which allows better optimization of membrane filtration processes.
Keywords: *filtration; parallel computing; optimization*

1. Filtration Model

The lattice Boltzmann method(LBM) is used to solve the Navier Stokes equations for fluid and convection diffusion equation for solute particles[1]. The general formulation of LBM is as follow.

$$f_\alpha(x_i + e_\alpha, t+1) - f_\alpha(x_i, t) = \Omega(f)$$

The left hand side represents the streaming step as each partial distribution function(PDF) shifts to a neighbour cell. The right hand side represents the collision step, where the collision operator rebalances each PDF to simulate change in direction due to particle collision. The details of the collision operator depend on the type of flow. For fluid flow we adopt the multiple relaxation time method by d'Humieres et al[2], for porous cells, such as membranes and cake formation, we adopt the modified BGK operator by Freed[3], and for solute particles the BGK operator is applied with variables altered to suit the convection diffusion equation.

To couple the Navier Stokes equations and convection diffusion equation the viscosity and diffusion coefficient are subject to changes according to the local concentration and shear rate.

When the local concentration of particles reaches a pre-set value, a lattice cell is considered to be a cake layer and modelled as a porous cell.

2. Code Restructuring

The first and foremost problem to identify is the dependencies of the simulation, as parallel computing involves multiple computational units it is vital to assign workload that does not clash with each other.

The collision step in LBM is completely local, as it only uses local information. Different processors can work on different lattice cells without interference. However at a processor level, reordering instructions can also improve performance, this is achieved by fully utilising all pipelines available. By calculating variables such as density, momentum and concentration at the beginning of a time step, most calculations in the collision step can be issued independent of each other, thus able to use all available pipelines and achieve a higher performance. Below is a comparison between the original and revised algorithm.

Solve fluid collision	Compute concentration and update viscosity
Compute macroscopic variables	Compute macroscopic variables
Solve solute collision	Solve fluid collision
Compute concentration and update viscosity	Solve solute collision
Algorithm 1: Original code	Algorithm 2: Revised version

In algorithm 1, each section of calculation is dependent on the previous section, which means there are three bottlenecks that the processor cannot proceed before every current workload is completed. In the updated version (Algorithm 2) calculations are independent of the last section of code, which means that as soon as a pipeline becomes free, it can carry out the next set of calculations instead of waiting for all other pipelines.

In a naive approach to perform the streaming step, one would shift the data to the neighbouring cells. This means reading data from the original location and saving to a new location, altering the value that was stored there. It requires a specific shifting order to ensure data correctness, and prohibits proper parallelisation. Moreover streaming is no more than a read and write operation to the processor, which is also exactly what the collision step has to do. By combing the two steps together only half as many memory transactions are required.

To accommodate these issues we adopt the AA pattern by Bailey et al.[4] There are two schemes for odd and even time steps. Figure 1 shows the memory operations for the cell in the centre. In odd time steps data is read as normal (first from left), but the PDFs are stored in the memory space of the opposite direction PDF (second from left). Even time steps perform a pull - collision - push, reading data from the neighbouring cells(second from right) and then storing similar to the previous step, swapping back to the natural memory space in the neighbours (first from right). The memory operations of this scheme is then completely local for each cell at each time step, since the processor read and write at the same location, and is inherently parallel.

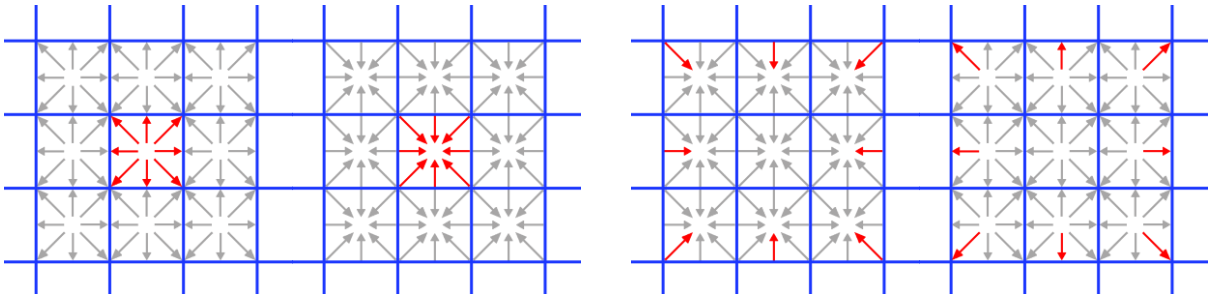

Figure 1: Read and write operations with AA pattern. Left: Odd time steps. Right: Even time steps.

The restructuring on the previous code has improved the performance by a factor of two.

3. Grid Division

For a cluster of CPUs or GPUs it is essential to divide the grid so that each CPU or GPU have equal workload and know which processors to communicate with. The current work concerns 2D problems and there are two different methods on grid division; dividing the grid along the longest axis, or dividing the grid along both axes. The former method is simpler and each processor has up to two neighbours, but has a longer boundary, the latter attempts to divide the grid into squares, thus having a smaller boundary but can have up to eight neighbours.

In GPU programming the simple grid division is clearly superior, due to coalesced memory access. On the other hand CPUs are less reliant on memory access pattern and a test was conducted to find out the differences.

Figure 2 shows the performance of 60 CPU cores on filtration simulation of various gird sizes, the performance in presented in terms of million lattice update per second (MLUPS), given by

$$\text{MLUPS} = \text{number of cells/time taken for a step}$$

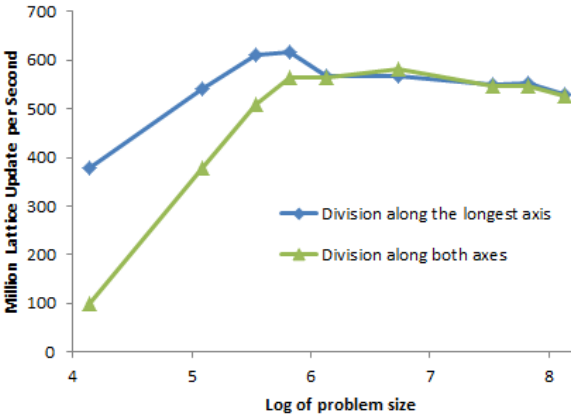

Figure 2: Performance of 60 CPU cores, using different grid division methods

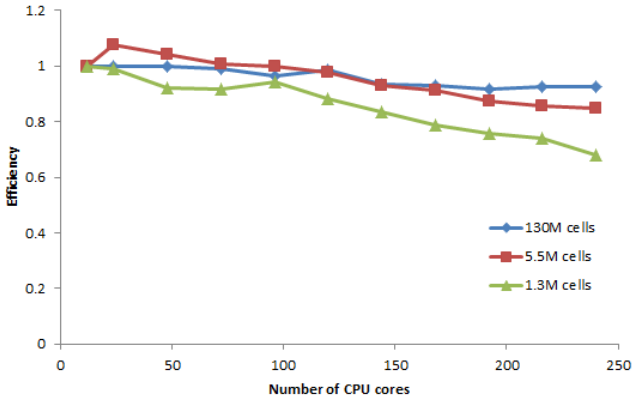

Figure 3: Efficiency of CPU cores on 3 grid sizes

The simple approach performs much better on small grids due to smaller communication overheads, and on large grids where the complex method communicates faster, the differences are minimal since calculations took much more time compare to communication. This shows that the simple grid division is better on CPU clusters as well.

4. Results and Discussion

Results on CPUs

Table 1 shows the speed up when fully utilising an 8-core CPU instead of just 1 core, the grids tested ranged from small 2D problem to large 3D problem. Up to 7.49 times was achieved which is equivalent to 93.75% efficiency.

Table 1: Factor of speed up on an 8-core CPU

Number of cells	13,680	123,120	1,368,000	5,472,000	34,200,000	136,800,000
Speed up	5.78	7.29	7.05	7.16	7.49	7.39

On a cluster of CPUs we tested the larger grids due to the scale of computing power when compared to one CPU. Figure 3 shows the efficiency of up to 240 CPU cores, which is over 85% and 92% for the 5.5million cells and 130million cells grid, whereas the 1.3million cells grid scale well with over 80% efficiency until over 140 cores were deployed. This is because the time taken for each time step falls under 1 millisecond, and the communication time becomes significant.

Bottleneck on CPUs:
While many other LBM simulations are described as memory bound, the current model includes coupling the fluid and solute particles, which introduced a long dependency chain and doubles the amount of calculations. Thus to a certain degree it is compute bound, which is why our multicore code is able to scale well with additional processors. However as it is necessary to inspect whether a cell is a fluid cell or porous cell, it requires reading information before any calculations can begin, which incur a section of code that is memory bound. This can be observed in Figure 2 as some of the smaller grids have the highest performance due to data fitting in the CPU cache and has a higher bandwidth.

Results on GPUs:
The current code was tested on a Titan Z and Quadro K5200. The Titan Z is a dual GPU with 5760 CUDA cores and 1:3 ratio of double precision units to single precision units. The Quadro K5200 has 2304 CUDA cores and 1:24 ratio of double to single units. Due to the difference in double precision capability, our code is compute bound on the Quadro and memory bound on the Titan. Kernels are launched with blocks of 512 threads, which allow each thread to have up to 128 registers and give optimal performance.

Table 2 shows the speed up compared to a sequential code. The Quadro K5200 GPU has a much lower performance at all stages than a single GPU on Titan Z due the lack of computing power. When both GPUs on the Titan Z are used, the communication between the two devices incurs a ~0.35ms penalty between time steps, which makes small grids very slow to process. On the larger grids the Titan Z is able to perform 72.7 times faster than a sequential code.

Table 2: Factor of speed up on two GPUs

Number of cells	13,680	123,120	1,368,000	5,472,000	34,200,000
Quadro K5200	9.2	17.5	19.1	20.1	20.4
Titan Z (1 GPU)	24.5	34.9	37.8	38.8	38.7
Titan Z (2 GPU)	1.8	40.1	55.1	70.5	72.7

5. Conclusion

An existing filtration simulation code was rewritten to make use of multiple processing units available in modern computer structures. The programmes are able to scale efficiently with increasing number of processors and provide performance that enable the simulation of industrial problems in an acceptable time scale.

References

[1] M. Paipuri, S.H. Kim, O. Hassan, N. Hilal and K. Morgan, Numerical modelling of concentration polarisation and cake formation in membrane filtration processes, *Desalination* 365 (2015), 151 – 159.

[2] D. d'Humieres, B. D. Shizgal, D. P. Weaver, Generalized lattice Boltzmann equations. In Rarefied gas dynamics: theory and simulations, Prog. Aeronaut. Astronaut. 159 (1992) 450 - 458.

[3] D. M. Freed, International Journal of Modern Physics C 9 (8)(1998), 1491 - 1503.

[4] P. Bailey, J. Myre, S.D.C. Walsh, D.J. Lilja, M.O. Saar, Accelerating lattice Boltzmann fluid flow simulations using graphics processors, *International Conference on Parallel Processing* (2009).