A Reduced-Storage Implementation of the Lattice Boltzmann Equation

R. Argentini¹, A.F. Bakker² and C.P. Lowe¹

¹ University of Amsterdam, Department of Chemical Engineering, Nieuwe Achtergracht 166,
1018 VW Amsterdam, The Netherlands

² Delft University of Technology, Faculty of Applied Sciences, Lorentzweg 1, 2628 CJ Delft,
The Netherlands

Abstract. Simulation programs using the Lattice Boltzmann Equation are limited in the range of problems they can address by memory requirements. We describe an implementation scheme that reduces memory usage by up to 78%. The performance of both the simple and the memory-reduced implementation are compared. Since memory access is the determinant step in the speed of execution of memory intensive applications, we look at the cache memory utilization in both the storage reduced version and in the simple reference implementation. From this we conclude that the proposed method does not degrade the performance relative to a simple implementation.

1 Introduction

Historically, the Lattice Boltzmann method was derived as a method for simulating the dynamics of fluids based on discretizing the Boltzmann equation [1]. At its simplest, the resulting algorithm can be regarded merely as a local second order discretization of the time-dependent non-linear Navier-Stokes equations (the differential equations that describe the dynamics of a Newtonian viscous fluid). As such, the Lattice Boltzmann method can be used to study a large number of phenomena involving fluid dynamics. A few diverse examples would be simulations of colloidal suspensions [2], the growth of coral formations [3] and flow through porous media [4]. The approach has also been applied in creative ways to non-fluid problems such as microwave propagation [5].

The computational cost involved in performing Lattice Boltzmann simulations [6], or any other fluid dynamics calculation, depends on the size of the simulation system involved and on the length of the simulation itself. Generally speaking, the size of the system determines the accuracy of the results, whereas the number of time-steps represent the time evolution of the physical quantities. Solving real-world problems with computational methods often requires operating on very large systems in order to correctly resolve complex geometries, or to produce results that are accurate and free of unwanted finite size effects. The problem with simulating large systems is of course that they are computationally very demanding, both in terms of processing time and of consumption of computer memory. In fact, it may not be possible to fit a complete representation of the problem space into the available computer memory. The quickest solution is to upgrade the machine's memory. As this depends on a number of external

factors, it might or might not be possible. While advances in out-of-core computation have been made in several disciplines including mathematics [7] and astrophysics [8], simply paging to bulk storage through the operating systems' virtual memory subsystem (known as "swapping") comes at the price of drastically slowing down the simulation's performance. This is generally not a good idea, as the simulation is already pushing the limits of the user's resources and patience.

By critically examining the equations that define the algorithm, we have devised a method of reducing the memory requirements without compromising the accuracy of the results or the speed of execution. With this approach we can now simulate problem spaces that are approximately four times larger, bringing more phenomena within the scope of this simulation technique. Alternatively, a more accurate representation of a given problem can be simulated within the constraints of fixed computational resources.

Fig. 1. A schematic representation of a cell of the face centered hyper-cubic lattice. Travel between cells is allowed only in the six (100) and the twelve (110) directions, for a total of eighteen possible displacement vectors.

2 Overview of the Lattice Boltzmann Method

The Lattice Boltzmann method operates, as the name suggests, by representing the fluid as discrete distributions of particles located on a lattice. In discrete time-steps particles are allowed to travel from one lattice site to another lattice site. This discretization of both space and time implies that the velocities of the particles are restricted to a discrete set as well.

The Lattice Boltzmann method can be viewed as a pre-averaged version of a lattice gas cellular automata [9]. This means that the propagation and collision rules are formulated not in terms of *particles* but in terms of *particle densities* $n_i(\mathbf{r}, t)$ that describe

the number of particles at lattice site $\mathbf{r}$ at time t with velocity $\mathbf{c}_i$. Quantities such as the mass density ρ , the momentum density $\mathbf{j}$ and the momentum flux Π are the moments of $n_i(\mathbf{r}, t)$:

$$\rho = \sum_i n_i, \quad \mathbf{j} = \sum_i n_i \mathbf{c}_i, \quad \Pi = \sum_i n_i \mathbf{c}_i \mathbf{c}_i \quad (1)$$

All quantities will be expressed in *lattice units*, where the distance between two nearest neighbors on the lattice and the time that it takes for a particle to travel between two adjoining lattice sites are equal to unity.

The time evolution of the distribution functions n_i is described by a discrete version of the Boltzmann equation [10]

$$n_i(\mathbf{r} + \mathbf{c}_i, t + 1) = n_i(\mathbf{r}, t) + \Delta_i(\mathbf{r}, t) \quad (2)$$

where Δ_i is the change in n_i due to collisions between particles that take place at the lattice sites and is a part of the *collision operator* $\Delta(n)$

There are now two distinct approaches to constructing the collision operator. The simplest is to use a BGK approximation [11] in which the density distribution functions n_i are relaxed toward their equilibrium values at a rate determined by a single parameter τ [12]

$$\Delta(n) = -\frac{n_i - n_i^{eq}}{\tau} \quad (3)$$

The single relaxation time constant τ can in turn be related to the viscosity of the fluid.

A slightly more involved (but actually more rigorously correct from the point of recovering Navier-Stokes behaviour) is the approach described by Ladd. Here the collision operator is constructed on the basis of its effects on the moments of the particle distribution. In practice it is often useful to linearize the collision operator $\Delta(n)$ about the local equilibrium n^{eq}

$$\Delta_i(n) = \Delta_i(n^{eq}) + \sum_j \mathcal{L}_{ij} (n_j - n_j^{eq}) \quad (4)$$

The actual form of $\mathcal{L}_{ij}$ can be derived from conservation of mass and momentum and by the requirement that the stress relaxes isotropically toward its equilibrium value (to ensure isotropic transport coefficients) [13]. In practice, when performing the simulation utilizing this approach, first the density, momentum density and momentum flux are evaluated. The momentum flux Π is then updated according to

$$\Pi'_{\alpha\beta} = \Pi^{eq}_{\alpha\beta} + (1 + \lambda) \left(\bar{\Pi}_{\alpha\beta} - \bar{\Pi}^{eq}_{\alpha\beta} \right) + \frac{1}{3} (1 + \lambda_b) (\Pi_{\gamma\gamma} - \Pi^{eq}_{\gamma\gamma}) \delta_{\alpha\beta} \quad (5)$$

where the eigenvalues of the collision operator λ and λ_b are related to the shear and bulk viscosity respectively through

$$\eta = -\frac{1}{6} \rho (2/\lambda + 1) \quad (6)$$

Having done so the post-collisional distribution is computed using

$$n_i + \Delta_i(n) = a_0^{c_i} \rho + a_1^{c_i} \mathbf{j}_\alpha \mathbf{c}_{i\alpha} + a_2^{c_i} \Pi'_{\alpha\beta} \mathbf{c}_{i\alpha} \mathbf{c}_{i\beta} + a_3^{c_i} (\Pi'_{\alpha\alpha} - 3\rho c_s^2) \quad (7)$$

Fig. 2. We initialize the density distribution buffer by calculating the post-collisional density distributions of the nodes in the first layer using either (7) or (9). We store the result in the sub-buffer named “current”. We also compute the post-collisional distribution densities of the neighboring layers in a similar fashion, and store them into the sub-buffers “top” and “bottom”. The node labeled “Layer 0” is a fictive layer, and the value of the density distribution is dictated by the boundary conditions. For example if periodic boundary conditions are used the moments of the bottom-most layer (N) would be used. When employing parallel simulation techniques one would use the bottom layer of the simulation box on top of the current one. This is commonly known as a “halo buffer”.

For our purposes here, the important point is that post-collisional distribution $n_i + \Delta_i(n)$ can be written in terms of the moments of the pre-collisional distribution.

We perform our computations using a cubic lattice where only propagation in the six (001) and in the twelve (011) directions is allowed. This is known as a face-centered hyper-cubic (FCHC) lattice (figure 1.) We also set the speed of sound $c_s^2 = \frac{1}{2}$ to maximize the stability of the simulation with respect to variations in the flow velocity. The value of the coefficients $a_j^{c_i}$ is then

$$\begin{aligned} a_0^1 &= \frac{1}{12} & a_1^1 &= \frac{1}{6} & a_2^1 &= \frac{1}{4} & a_3^1 &= -\frac{1}{6} \\ a_0^{\sqrt{2}} &= \frac{1}{24} & a_1^{\sqrt{2}} &= \frac{1}{12} & a_2^{\sqrt{2}} &= \frac{1}{8} & a_3^{\sqrt{2}} &= \frac{1}{12} \end{aligned} \quad (8)$$

For the special case where the non-linear term in the equilibrium momentum flux tensor is neglected (so the Reynolds number is zero) and the viscosity $\nu = \frac{1}{6}$, equation 7 can be simplified to

$$n_i + \Delta_i(n) = a_0^{c_i} \rho + a_1^{c_i} \mathbf{j}_\alpha \mathbf{c}_{i\alpha} \quad (9)$$

thereby eliminating the dependency of the post-collisional distribution on the momentum flux Π . Note that because the Reynolds number is, in this case, by definition equal to zero, this will only generally apply for flows on a very small scale. It will not apply for turbulence. Nonetheless there are plenty of interesting problems where this is the case, on a cellular scale for example, so this is not an irrelevant limit. It should also be noted that if the Reynolds number is zero, the kinematic viscosity ν becomes a parameter that only scales time in the problem, so the restriction to one particular viscosity is not very restrictive.

3 Reducing Memory Requirements

At a conceptual level, one can think of a Lattice Boltzmann simulation as the act of applying two operations on the computational domain. The first operation is *propagation*. During this operation particles are allowed to travel between nodes, as dictated by their velocity c_i . The other operation consists of simulating the effects that the particles at a certain node have on each other by applying the collision operator. These operations give rise to specific states of the lattice occupation that we will call the *pre-collisional* and *post-collisional* distributions respectively.

Fig. 3. We can now proceed to the propagation step. Looping over all nodes in “current” we calculate the moments of the density distribution by considering the densities that *flow toward* the node. The resulting moments are stored back into the Moments buffer.

Based on the propagation equation (equation 2) one would imagine that is necessary to know all the values of the single particle distribution functions n_i in order to update the system over one time-step. This is indeed one possible approach which we will subsequently refer to as the “simple” method. However, from equations 7 and 9, it is readily apparent that the post-collisional distribution actually depends only on the moments of n_i rather than on all the components of n_i . Therefore storing the moments of the pre-collisional distribution is actually sufficient to be able to re-create the entire post-collisional distribution at a later time by simply applying the collision operator. Note that this methodology cannot (transparently at least) be applied for the Lattice BGK approach. Using this method it is implicit that the values of n_i are required (see eq. 3.)

The propagation operation operates on the actual values of n_i . Since particles are only allowed to travel between neighboring nodes, this operation is of a very localized nature. Evaluation and storage of n_i for the entire domain is therefore not necessary. When using a FCHC lattice one only needs to calculate one component of n_i in the 18 neighboring nodes to compute the pre-collisional distribution.

In practice it is often more convenient to work layers. We take a computational domain D in the form of a rectangular parallelepiped, divide it in layers of nodes along the

longest dimension l and number the layers from 1 to N in the direction of increasing l . Now suppose we want to perform the propagation operation on the nodes that make up layer n . The first step is to obtain the values of n_i of the pre-propagation (i.e. post-collisional) distribution for the layers $n - 1$, n and $n + 1$ by applying collision operator (7) or (9) to the moments. The results are stored in a temporary memory buffer. We now

Fig. 4. Since the “current” of “Layer n ” is the “top” of “Layer $n+1$ ”, and the “bottom” of “Layer n ” is the “current” of “Layer $n+1$ ”, to move processing to the next layer we only need to move density distributions of these layers (or their pointers) to the appropriate place. Finally we calculate the post-collisional density distribution of “Layer $n+1$ ” and store the result in “bottom”. We have now placed ourselves in a situation similar to that of figure 3, and we can iterate this procedure until all layers of the computational domain have been processed.

have all the information we need to calculate the post-propagation (i.e pre-collisional) distribution of the nodes in layer n , but instead of simply storing it in the space reserved for n in the temporary storage, we accumulate it directly into the moments buffer by applying (1). The reason for this is twofold. It avoids the need to iterate through the data one more time, thus lessening the possibility of cache misses and improving performance. More importantly, it leaves the pre-propagation distribution of layer n intact, allowing the use of a “sliding window” to compute the moments of the post-propagation distribution in the rest of the domain. All we need to do is replace the pre-propagation distribution of layer $n - 1$ with that of layer $n + 2$, and perform the operation outlined above for layer $n + 1$.

We start at $n = 0$, with $n - 1$ dictated by the boundary conditions. By repeatedly applying the algorithm outlined above we can slide the window all the way to $n = N$, with $n+1$ likewise dictated by the boundary conditions. The total memory requirements in number of floating point numbers for a cubic system with dimensions $N \times N \times N$ are for the case of $\nu = \frac{1}{6}$

$$M_{\nu=\frac{1}{6}} = N^3 + 3N^3 + 3 * 18N^2 = 4N^3 + 54N^2 \quad (10)$$

and for the case of $\nu \neq \frac{1}{6}$

$$M_{\nu \neq \frac{1}{6}} = N^3 + 3N^3 + 5N^3 + 3 * 18N^2 = 9N^3 + 54N^2 \quad (11)$$

Fig. 5. Memory consumption as function of the size of the computational domain for the simple (solid) general reduced-storage (dashed) and special $\nu = \frac{1}{6}$ (dotted) implementations. The dot-dashed line represents the amount of memory available on a typical small workstation.

In contrast a simple implementation would require the storage of $M_{\text{simp}} = 18N^3$ floating point numbers. Figure 5 shows that, as N gets larger, memory savings become more and more significant. For the case of $\nu = \frac{1}{6}$ memory requirements drop by up to 78% for large values of N .

For the sake of completeness it might be worthwhile noting that the algorithm proposed here performs the same calculations as the simple implementation, albeit on data organized in a different fashion. It therefore yields exactly the same results.

4 Performance

Memory access speeds are significantly lower than the capacity of modern processing units to perform operations. Therefore main memory is accessed through at least two smaller but faster *cache memories* [14], usually called L1 and L2.

The L1 cache operates at or near processor speed, and the processor can fetch values from it with no extra overhead. If the program requires a value that is not stored in the L1 cache a *cache miss* occurs. The value must then be fetched from the (slower) L2 cache. This results in 10 to 20 cycles that the processor must spend waiting for data. During that time it is unable to do any useful work. Sometimes the requested value will not be present in the L2 cache and an access to main memory will be required. The

penalty associated with fetching data directly from main memory is in the order of 100 cycles.

In light of this, it is imperative to implement the simulation algorithm in such a fashion that cache misses are minimized by properly ordering memory accesses and by attempting to keep the number of memory locations in active use as small as possible. The programmer has a number of tools at his disposal for monitoring memory access performance, such as hardware performance counters integrated in the processor [15] and cache simulators [16].

Fig. 6. L2 cache memory utilization for the simple and reduced-storage implementation for different sizes of the computational domain, obtained using the Valgrind [16] simulator and memory debugger. We observe that for system sizes up to 50^3 the reduced-storage implementation is able to fit a considerable portion of the domain into the L2 cache. The simple implementation does not appear to benefit from caching for systems of sizes 30^3 or larger.

Figure 6 gives an overview of L2 cache memory utilization for the simple and reduced-storage implementation for different sizes of the computational domain, obtained using a simulator. The simulator was instructed to simulate a 64KB, 2-way associative L1 data cache and a 256KB 8-way L2 cache, both with 64 byte cache lines. The results of L1 utilization are all but irrelevant for systems of useful dimensions, and have therefore been omitted from the figure.

It is readily apparent that for large system sizes both methods suffer approximately the same number of cache misses per processed simulation node. This suggests that the extra bookkeeping involved in the storage-reduced implementation does not result in additional uncached memory accesses and thus in a large performance degradation. We also learn that the storage-reduced implementation is able to make good use of the L2 cache for system sizes under 50^3 , and could in such cases possibly even outperform the simple implementation.

Fig. 7. Simulation speed in cell updates per second as function of the size of the computational domain for the simple (solid) and memory reduced $\nu = \frac{1}{6}$ (dashed) implementations. Simulating a cubic domain with more than 140 cells per side with the simple implementation requires more memory than is physically present in the machine, and processing slows down dramatically as the system resorts to paging. These results were obtained on an Athlon 1.4 GHz with 512 MB of memory.

Figure 7 shows the speed of execution of both the simple implementation and implementation with memory reduction. Note that simulating a cubic domain with more than 140 cells per side with the simple implementation requires more memory than is physically present in the machine. Due to the smaller memory footprint, the implementation with memory reduction is able to continue unhindered until approximately 220^3 .

In view of the results obtained from the cache memory simulator, it is not surprising that the memory optimized version performs marginally better than the simple implementation for small sizes of the computational domain. If the computational domain is smaller than 50^3 a large portion of the domain resides in the L2 cache, decreasing the amount of time the program has to wait to fetch data out of main memory.

In the intermediate regime, where the problem is too large to fit in cache, but still small enough to fit in main memory even in the unoptimized case, the performance of the two methods is roughly equal. This reflects the fact that both algorithms are executing the same operations, and that the overhead incurred in the extra bookkeeping for layered processing is small compared to the amount of work needed to perform the actual simulation.

5 Conclusions

We have developed a technique for reducing the memory requirements of a Lattice Boltzmann simulation. By applying this technique to a sufficiently large computational domain, memory savings can be realized from 44% in the general case up to 78% in the special case where the viscosity $\nu = \frac{1}{6}$. Reducing the size of the memory representation

of the problem space makes it possible to work on problems that might otherwise be inaccessible due to hardware limitations. The method appears not to incur any performance penalties while operating in the intermediate regime, and can even have slightly beneficial effects for the speed of the simulation when the problem space is small and a greater portion of the domain fits entirely into fast cache memory. It should be noted, however, that any such improvement is marginal and the main advantage of our approach is reducing memory usage rather than minimizing execution time. It is possible that different storage schemes or data-prefetching techniques may minimize cache misses and improve the performance but we have not as yet investigated this. Because of this it may be of use for anyone performing simulations with the Lattice Boltzmann equation within the context of fixed computational resources.

References

1. G.R. McNamara and G. Zanetti. "Use of the Boltzmann equation to simulate lattice-gas automata". *Phys. Rev. Lett.*, 61:2332, 1988.
2. A.J.C. Ladd, H.Gang, J.X. Zhu and D.A. Weitz. "Time-dependent collective diffusion of colloidal particles". *Phys. Rev. Lett.*, 74:389, 1995.
3. J.A. Kaandorp, C.P. Lowe, D. Frenkel and P.M.A. Slood. "The effect of nutrient diffusion and flow on coral morphology". *Phys. Rev. Lett.*, 77:2328, 1996.
4. A. Heijs. "3D Simulation and Visualization studies of flow in Porous Media". PhD thesis, Delft University of Technology, 2001.
5. B. Chopard, P.O. Luthi and J-F. Wagen. "A lattice Boltzmann method for wave propagation in urban microcells". *IEEE Proceedings - Microwaves, Antennas and Propagation*, 144:251–255, 1997.
6. S. Succi. *The Lattice Boltzmann equation*. Oxford University Press, Walton Street, Oxford OX2 6DP, 2001.
7. E. Caron and G. Utard. "on the performance of parallel factorization of out-of-core matrices". *Parallel Computing*, 2002.
8. J. Salmon, M.S. Warren. "Parallel Out-of-core Methods for N-body Simulation". *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.
9. U. Frisch, B. Hasslacher and Y. Pomeau. "Lattice-gas automata for the Navier-Stokes equation". *Phys. Rev. Lett.*, 56:1505–1508, 1986.
10. U. Frisch, D. d'Humieres, B. Hasslacher, P. Lallemand, Y. Pomeau and J-P. Rivet. "Lattice-gas hydrodynamics in two and three dimensions". *Complex Systems*, 1:649, 1987.
11. P. Bhatnagar, E. Gross, and M. Krook. "A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems.". *Physical Review*, 94 (3):511–525, 1954.
12. Y. Qian and S.A. Orszag. "Lattice BGK models for the Navier-Stokes equation". *Europhys. Letters*, 17:479–484, 1993.
13. A.J.C. Ladd. "Numerical simulations of particulate suspensions via a discretized Boltzmann-equation. 1. Theoretical foundation". *J. Fluid Mech.*, 271:285–309, 1994.
14. S.A. Ward and Jr. R.H. Halstead. *Computation Structures*. The MIT Press, Cambridge Massachussets, USA, 1990.
15. Intel Corporation. "Pentium(R) Processor Family Developer's Manual", 1997. <http://developer.intel.com/design/intarch/manuals/24142805.pdf>.
16. J. Seward and N. Nethercote. Valgrind open source memory debugger, 2002. <http://developer.kde.org/sewardj/>.