ENHANCED ASYNCHRONY IN THE VECTORIZED CONEFOLD ALGORITHM FOR FLUID DYNAMICS MODELLING

Perepelkina A. Ph. D., Levchenko V. Ph.D.
Keldysh Institute of Applied Mathematics, Miusskaya sq. 4, Moscow, Russia
Email: mogmi@narod.ru, lev@keldysh.ru

Abstract: *The application of a rigorous CFD method and an all-encompassing algorithmic performance optimization method can make possible the CFD simulation of the extremely large-scale problems, which allows simulation of either larger systems, or more detailed simulation of systems that are already simulated. The CFD code has to show both efficient one-node performance and excellent parallel scaling. The record breaking performance on one node has been achieved before with application of the LRnLA algorithm and making use of many core parallelism as well as the vectorization. In the current work, the algorithm is extended for many-node parallelism. The algorithms is characterized by high parallelization degree, small number of node communication events, and may be concisely described and programmed on the base of the previously implemented one-node solution, which is a rare feature among the algorithms with temporal blocking in all four of the spatial and time dimensions.*

Keywords: LATTICE BOLTZMANN METHOD, PARALLEL ALGORITHMS, LRNLA, CONEFOLD, CFD

1. Introduction

The mathematical modelling of fluids is used for industrial design problems, disaster prevention, exploration for oil and gas, as well as in the medical applications. The CFD problems are computationally heavy, and comprise a large portion of the supercomputer load. To make the large-scale modelling cheaper, accessible to more scientists, and to make extreme scale modelling possible, we develop algorithms that raise the efficiency of the parallel implementation of the numerical schemes higher than the memory bound limit.

Among the CFD schemes the Lattice Boltzmann Method [14] has the advantage of high stencil locality. It is highly parallelizable, and the speedup from using the hybrid computers is achieved by many authors. The method has its range of stability, and there are extensions that allow for more stability in simulation of high Reynolds numbers. However, the extensions of the method may complicate the simulation so that it may be comparable in amount of computation to the Navier-Stokes discretization schemes. We promote the other way to extend the range of possible applications: by making the computer implementation of a simple scheme more efficient, larger meshes may be simulated in reasonable time. This way, the robustness of the method is gained by highly detailed mesh..

Indeed, high performance LBM codes [1,2,6,7,8,12,13] use the most basic variations of the method. In our work we achieved the record breaking performance on multi-core CPU [6,10] and high-end GPU [6,7]. These results were obtained by applying the LRnLA algorithms [4,5] that allow traversal in both space and time to enhance the locality of data access and take advantage of the computer memory hierarchy to gain more calculation performance. The approach of space-time decomposition of the problem has been used in LBM codes by other authors to conceal data copy in parallel simulation [13] and to overcome some of the memory bottlenecks [8]. In CFD LRnLA algorithms are also applied for the RKDG method [3].

In this paper we extend our previous algorithm to make the multi-node simulation possible.

2. Methods

In LBM [14], the simulation domain is split into Nx×Ny×Nz cubic cells. In each cell, the probability distribution function is known for a set of discrete velocities $\vec{c}_{ijk}$. The specific method is denoted by a word like D3Q19, where the first number is the dimensionality of the model and the second number is the number of velocities. Discrete velocities are chosen as vectors that point from the center of the cell to the centers of its neighbors, and a zero velocity. In D3Q27, there is a set of vectors that point to each cell in a 3×3×3 cube. In D3Q19, the longest vectors of D3Q27 are pruned.

For each velocity the update rule for its Distribution Function (DF) is split into two sub-steps: the streaming step $f_{ijk}(\vec{r}_{ijk}, t+\Delta t) \leftarrow f_{ijk}(\vec{r}_{000}, t)$, and the collision step after: $f_{ijk} \leftarrow f_{ijk} - (f_{ijk} - f_{ijk}^{eq})/\tau$; i, j, k = -1, 0, 1 while $i^2+j^2+k^2<3$for D3Q19.

Streaming copies the f_{ijk} from cell with coordinates r_{000} to the cell with the relative position $\vec{r}_{ijk} = \vec{r}_{000} + \vec{c}_{ijk}\Delta t$, $\vec{c}_{ijk} = (i, j, k)$. The collision operates with the DF in the same cell. The expression for the equilibrium DF f_{ijk}^{eq} is taken as the most commonly used second-order polynomial in $\vec{u} = \sum_{ijk} f_{ijk}\vec{c}_{ijk} / \sum_{ijk} f_{ijk}$ [14] to make the performance comparison easier, but any expression that operates on the data inside one LBM cell may be used in the current implementation.

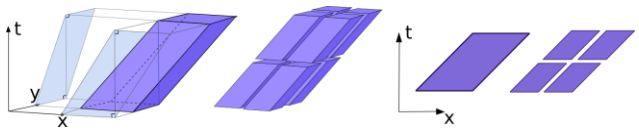

Fig. 1 *ConeFold algorithm in 2D1T and 1D1T*

For the implementation, we start from our previous work documented in [10]. The reader may refer to the texts for information of algorithm construction, data structure, details on the vectorization method. These are summarized below.

2.1 ConeFold

The algorithm operates recursively on a Z-curve array, which is a cube with linear size of $N=2^{MaxRank}$, where MaxRank is an integer number. Between the synchronization steps in time t=0 and t=N the dependency graph is subdivided recursively (Fig. 1) until an elementary update of one cell.

The procedure is implemented with recursive templates in C++. There are special cases for the inside of the domain, left and right boundaries, the decomposition in 1D1T is shown in Fig. 2 [11]. For 2D1T and 3D1T the treatment of all corners is necessary, and the coding similar to a direct product, since the description of ConeFold may be split by coordinates. For example, in 3D1T at x=N, y=N, z=0 the code is XXI.

In 1D1T case, at maximal rank, two ConeFold should be executed: X and I. Each of them will recursively call ConeFolds of smaller rank (Fig. 2). In 3D1T, after XXX, three ConeFolds may be executed in parallel (XXI, XIX, IXX), as well as the next three (IIX, IXI,XII). The last one is III.

Fig. 2 *ConeFold decomposition and codes near the boundaries in 1D1T*

2.2 Streaming algorithm

The smallest ConeFold is an LRnLA cell. Its base is one data structure cell. Two of the streaming algorithms allow for only one LBM node to be put in this structure: EsoTwist [1] and the special swap algorithm used in our previous work [10]. Here we have implemented EsoTwist for further comparison of the methods. In EsoTwist, in the LRnLA cell the data that is saved is put in place of the data that was read for its execution. This prevents data race condition when parallelism is implemented with stepwise algorithms. In ConeFold, this advantage is not used. However, at smaller scales, this may lead to better data locality.

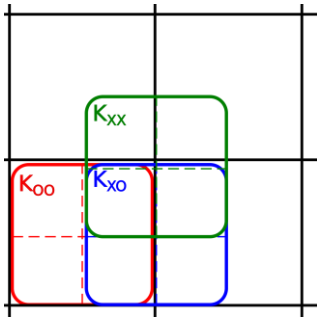

Fig. 3 *Implementation of ConeTur with vectorized ConeFold*

2.3 AVX vectorization

The AVX2/AVX512 vectorization is implemented by putting 8 float/double values into each DF in a cell. At the same time, the size of the domain is doubled in each direction.

To return to 1D1T illustration, let the vector length be equal to 2, not 8. These two values are from cells *ix* and *2N-ix-1* in the domain. It may be visualized as if the *2N* cell domain is 'folded', the ends are 'glued' and we get a 'ring' of cells. First N cells are in the natural order, the next N cells are mirrored. The LBM scheme is also mirrored. The ConeFold is executed on a cube of N cells, which contain the vectors of natural and mirrored data. This way, the calculation inside the domain with scalars for the N size domain and the calculation in the with vectors for the 2N domain are indistinguishable. The mirroring is implemented only by introducing {1,-1} constant vectors into the numerical scheme. At X and I ConeFold, the mirrored and natural domains are linked by the vector shuffle operations. This is generalized to 3D, where the vector length is 8, the domain size is 2N×2N×2N, and some areas are mirrored by several axes.

This kind of implementation of periodic boundary with wavefront blocking is also suggested in [9].

3. Multi-node parallelism

In the current work we explore the ability to make a many-node implementation of the code.

There is a certain issue with multi-node parallelism of the 3D1T ConeFold, especially prominent when the number of nodes is high. Let us consider a simulation domain, made up of Bx×By×Bz cubes. The cubes may be passed to different nodes, and each node would start the ConeFold with MaxRank, treating the cube as a base. The maximal degree of parallelism can be estimated as a number of cubes on the 3D diagonal cross-section of the domain.

On the other hand, other types of space-time traversal algorithms allow more parallelism, such as diamond tiling [15] or ConeTur [4, 5], or earlier version of the LRnLA algorithms. In 2D1T subdivision, which we choose for the demonstration since it can illustrate some complexity of higher dimensions by using 3D shapes, the tiling is obtained with octahedra and two types of tetrahedra. In our work, the ConeFold is favoured since it is simpler for the programmer to write and for the compiler to optimize, and produces a clean and comprehensible code due to the use of the common recursive C++ templates. The number templates types is equal to 3^D, due to the fact that the special treatment is required for the boundary of the domain. If ConeTur is used, in 3D1T there are 2^D types of shapes, and each shape has to be specified in the variants that describe each of the 3^D-1 boundary types.

The use of 'folding' of the domain which has already helped with vectorization and application of the periodic boundaries can be used in this case as well, for the purpose of simplifying the implementation of ConeTur.

In 2D1T illustration (Fig. 3), which is easily generalized to 3D1T, let us take the area of 2×2 cubes K_{OO}, and fold them to make a vectorized cube K_V. If only the first ConeFold XX is executed on K_V, it is equivalent to the execution of the 2D1T pyramid on the K_{OO} base. The data from K_V is returned to the main array. Then 2×2 cubes K_{OX}, are folded into K_V, and ConeFold IX is executed on it. This fills in the tetrahedron between the pyramids in the X direction. The same is performed in the Y axis, and then for K_{XX}. At this point every cell is updated up to the synchronization instant.

Thus, ConeTur is executed by 'refolding' of the ConeFold. The synchronization between nodes, in case the pyramids and the tetrahedra are distributed between different nodes, is performed 2^D times per N time steps.

4. Performance analysis

We have implemented the described algorithm, namely, the 'unfolding' and 'folding' the data cubes. On one node, in the Bx×By×Bz cubes domain, one vectorized cube is formed, processed, and unfolded repeatedly to update all cubes on the node. At the boundary, in case there is a part of domain that is processed by another node, the necessary amount of data is sent by MPI. Otherwise, the one-node domain is treated as periodic.

The important metric for the performance analysis is the slowdown due to the vector copy operations. Thus, we have tested the code with and without the 'refolding' introduction and compared the results on different processors.

The test was performed on one node with N=256, Bx×Bx×Bx = 2×2×2, D3Q19. The performance is compared with the previously published results [10]. Since the results do not differ much, we conclude that the data copy, which is introduced with the new algorithm, does no significant impact on the performance.

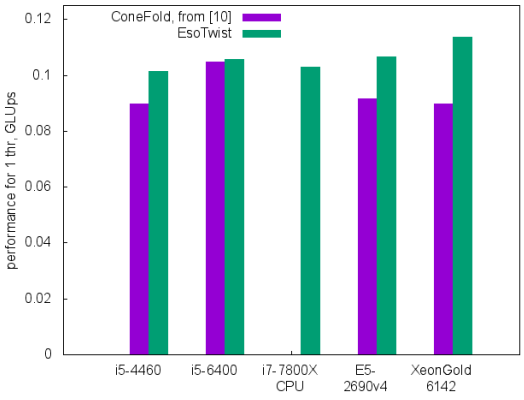

Fig. 4 *Performance comparison against the previous solution*

5. Conclusion

We have introduced the extension of the vectorized ConeFold algorithm for the supercomputer fluid simulation with the Lattice Boltzmann Method. The ConeTur algorithm, that has been difficult to implement in 3D1T before, has been implemented by reshuffling the data structure in the code based on ConeFold. This solution leads to a comprehensible code for fluid simulation, where the space-time decomposition is used for 3 levels of parallelism: vectorization, multi-core and multi-node.

We see that the introduction of the 'refolding' algorithm has not presented significant slowdown. The current results are even higher than the reported ones, which is probably due to small optimization of the code and compiler options.

The introduced algorithm may be applied to other hydrodynamic schemes with cube stencil, and for similar schemes of other numerical methods.

The work is supported by the Russian Science Foundation (project #18-71-10004).

6. Literature

1. Geier, M., Schönherr, M.: Esoteric twist: an efficient in-place streaming algorithmus for the lattice boltzmann method on massively parallel hardware. Computation 5(2), 19 (2017)

2. Godenschwager, C., et al.: A framework for hybrid parallel flow simulations with a trillion cells in complex geometries. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. p. 35. ACM (2013)

3. Korneev B., Levchenko V. Numerical simulation of increasing initial perturbations of a bubble in the bubble–shock interaction problem // Fluid Dynamics Research. – 2016. – V. 48. – No. 6. – P. 061412.

4. Levchenko, V. D. Asynchronous parallel algorithms as a way to archive effectiveness of computations. J. of Inf. Tech. and Comp. Systems 1 (2005): 68.

5. Levchenko V., Perepelkina A. Locally recursive non-locally asynchronous algorithms for stencil computation // Lobachevskii Journal of Mathematics. – 2018. – V. 39. – No. 4. – P. 552-561.

6. Levchenko, V., et al. GPU Implementation of ConeTorre Algorithm for Fluid Dynamics Simulation." International Conference on Parallel Computing Technologies. Springer, Cham, 2019.

7. Levchenko V., et al. LRnLA Lattice Boltzmann Method: A Performance Comparison of Implementations on GPU and CPU. In: Sokolinsky L., Zymbler M. (eds) Parallel Computational Technologies. PCT 2019. Communications in Computer and Information Science, vol 1063. Springer, Cham

8. Nguyen, A., et al.: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–13. IEEE (2010).

9. Osheim, Nissa, et al. "Smashing: Folding space to tile through time." International Workshop on Languages and Compilers for Parallel Computing. Springer, Berlin, Heidelberg, 2008.

10. Perepelkina, A., Levchenko, V.: LRnLA algorithm ConeFold with non-local vectorization for LBM implementation. Commun. Comput. Inf. Sci. 965, 101–113 (2019)

11. Perepelkina, A.Y., Levchenko, V.D., Goryachev, I.A.: Implementation of the kinetic plasma code with locally recursive non-locally asynchronous algorithms. In: Journal of Physics: Conference Series. vol. 510, p. 012042. IOP Publishing (2014)

12. Riesinger, C., Bakhtiari, A., Schreiber, M., Neumann, P., Bungartz, H.J.: A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters. Computation 5(4), 48 (2017)

13. Shimokawabe, T., Endo, T., Onodera, N., Aoki, T.: A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers. In: Cluster Computing (CLUSTER). pp. 525–529. IEEE (2017)

14. Succi, S.: The Lattice Boltzmann Equation: for Fluid Dynamics and Beyond. Oxford University Press, Oxford (2001)

15. Orozco, Daniel, and Guang Gao. Diamond tiling: A tiling framework for time-iterated scientific applications. CAPSL Technical Memo 091, 2009.