Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 994–1003

Procedia
Computer Science

International Conference on Computational Science, ICCS 2011

Optimization of Multi-Phase Compressible Lattice Boltzmann Codes on Massively Parallel Multi-Core Systems

Luca Biferale[a], Filippo Mantovani[b], Marcello Pivanti[c], Fabio Pozzati[c], Mauro Sbragaglia[a], Andrea Scagliarini[a], Sebastiano Fabio Schifano[c], Federico Toschi[d], Raffaele Tripiccione[c]

[a] *University of Tor Vergata and INFN, Roma, Italy*
[b] *Deutsches Elektronen Synchrotron (DESY), D-15738 Zeuthen, Germany*
[c] *University and INFN, Ferrara, I-44124 Ferrara, Italy*
[d] *Eindhoven University of Technology, The Netherlands and CNR-IAC, 00185 Rome, Italy*

Abstract

We develop a Lattice Boltzmann code for computational fluid-dynamics and optimize it for massively parallel systems based on multi-core processors. Our code describes 2D multi-phase compressible flows. We analyze the performance bottlenecks that we find as we gradually expose a larger fraction of the available parallelism, and derive appropriate solutions. We obtain a sustained performance for this ready-for-physics code that is a large fraction of peak. Our results can be easily applied to most present (or planned) HPC architectures, based on latest generation multi-core Intel processor architectures.

Keywords: Computational fluid-dynamics, Lattice Boltzmann methods, multi-core processors

1. Overview

Fluid-dynamics critically relies on computational techniques to compute reliable solutions to the highly non-linear equations of motion in regimes interesting for physics or engineering. Over the years, many different numerical approaches have been theoretically developed and implemented on state-of-the-art massively parallel computers.

The Lattice Boltzmann (LB) method is a flexible approach, able to cope with many different fluid equations (e.g., multiphase, multicomponent and thermal fluids) and to consider complex geometries or boundary conditions. LB builds on the fact that the details of the interaction among the fluid components at microscopic level do not change the structure of the equations of motion at the macroscopic level, but only modulate the values of their parameters. LB then describes on the computer some simple synthetic dynamics of fictitious particles that evolve explicitly in time and, appropriately averaged, provide the correct values of the macroscopic quantities of the flow; see [1] for a complete introduction.

The main advantage of LB schemes from the point of view of an efficient parallel implementation is that they are local (that is, they do not require the computation of non local fields, such as pressure). However, in recent years, the processing nodes themselves have included more and more parallel features, such as many-core structures and/or vectorized data paths: the challenge now rests in combining effectively inter-node and intra-node parallelism.

In this paper, we report on a high-efficiency implementation of LB on a massively parallel system whose compute nodes are themselves state-of-the-art multi-core CPUs. This paper builds on previous work that addressed the same

doi:10.1016/j.procs.2011.04.105

problem for a massively parallel architecture based on the IBM PowerXCell8i processor, and has been used for large-scale simulations of the Rayleigh-Taylor instability [2]. In this note we tailor our codes for state-of-the-art Intel architectures, and perform extensive tests on a recent massively parallel system (the AuroraScience machine) based on a large array of these processors connected by a toroidal 3D network. We make sure however that our results can be quickly ported to most recent parallel machines.

2. Lattice Boltzmann Methods

In this section we briefly describe the Lattice Boltzmann approach to the numerical solution of a class of Navier-Stokes equations that describe fluid flows. Full details can be found in [3, 4]. This computational method simulates the behavior of a compressible gas/fluid. The Thermal-Kinetic description of a compressible gas/fluid of variable density, ρ, local velocity $\boldsymbol{u}$, internal energy, $\mathcal{K}$ and subject to a local body force density, $\boldsymbol{g}$, is given by the following equations:

$$\partial_t \rho + \partial_i(\rho u_i) = 0 \tag{1}$$

$$\partial_t(\rho u_k) + \partial_i(P_{ik}) = \rho g_k \tag{2}$$

$$\partial_t \mathcal{K} + \frac{1}{2}\partial_i q_i = \rho g_i u_i \tag{3}$$

where P_{ik} and q_i are the momentum and energy fluxes.

In the continuum, one shows that it is possible to recover exactly these equations, starting from a continuum Boltzmann Equations and introducing a suitable shift of the velocity and temperature fields entering in the local equilibrium [3].

The discretized counterpart of the continuum description (that we use in this paper) uses a set of fields $f_l(x, t)$ associated to the so-called *populations*; the latter can be visualized as pseudo-particles moving in appropriate directions on a discrete mesh (see fig. 5). The master evolution equation in the discrete mesh is:

$$f_l(\boldsymbol{x} + \boldsymbol{c}_l \Delta t, t + \Delta t) - f_l(\boldsymbol{x}, t) = -\frac{\Delta t}{\tau}\left(f_l(\boldsymbol{x}, t) - f_l^{(eq)}\right)$$

where subscript l runs over the discrete set of velocities, $\boldsymbol{c}_l$ (see again fig. 5) and equilibrium is expressed in terms of hydrodynamical fields on the lattice, $f_l^{(eq)} = f_l^{(eq)}(\boldsymbol{x}, \rho, \bar{\boldsymbol{u}}, \bar{T})$. We remark that our code, that uses 37 populations (a so-called D2Q37 code), requires more memory and more flops than standard LB approaches (e.g., D2Q9 or D3Q19).

To first approximation, the macroscopic fields are defined in terms of the lattice Boltzmann populations: $\rho = \sum_l f_l$, $\rho\boldsymbol{u} = \sum_l \boldsymbol{c}_l f_l$, $D\rho T = \sum_l |\boldsymbol{c}_l - \boldsymbol{u}|^2 f_l$. When going into all mathematical details, one finds that shifts and renormalizations have to be applied to the averaged hydrodinamical quantities to correct for lattice discretization effects. After performing these manipulations, one recovers the correct thermo-hydrodynamical equations [3, 4]:

$$D_t \rho = -\rho \partial_i u_i^{(H)} \tag{4}$$

$$\rho D_t u_i^{(H)} = -\partial_i p - \rho g \delta_{i,2} + \nu \partial_{jj} u_i^{(H)} \tag{5}$$

$$\rho c_v D_t T^{(H)} + p \partial_i u_i^{(H)} = k \partial_{ii} T^{(H)} \tag{6}$$

where we have introduced the material derivative, $D_t = \partial_t + u_j^{(H)}\partial_j$, we have neglected viscous dissipation in the heat equation (usually small) and the superscript H denotes the lattice-corrected quantities; c_v is the specific heat at constant volume for an ideal gas $p = \rho T^{(H)}$, ν and k are the transport coefficients.

3. The AuroraScience Machine

The massively parallel AuroraScience machine [5], that we use for test and performance measurements, adopts latest generation multi-core processors and interconnects them with a custom 3D-torus network, that supports only nearest-neighbor data transfers with low communication latency. This system architecture has been recently adopted by the QPACE project [6, 7], based on the IBM PowerXCell8i processor.

The AuroraScience processing elements have two Intel X5680 CPUs(code-name Westmere), 12 GBytes of DDR3 RAM and a communication processor implemented on a *Field Programmable Gate Array* (FPGA). The CPUs run at a frequency of 3.33 GHz, include six cores and 12 MBytes of L3-level shared cache, share the memory and operate as a Symmetric Multiprocessor, running the Linux operating system. Peak performance is around 160 Gflops, double precision.

The network processor has 6 links (10 Gbit/sec each, full-duplex) supporting the toroidal mesh. It is based on an open source design [9], which includes the processor interface, and the logic to send and receive data on the six communication links. The CPUs send messages by writing one or more 128 Byte data-packet to one of the six queues, associated to the six directions. As soon as the sending queue is not empty, the associated TX logic starts sending the packet over the links. On the receiving side the RX logic receives the packets, checks for errors, and if a credit has been provided by the receiving application, writes the packet to the memory via a DMA operation. For our present application, only a subset of these features is really needed.

Blocks of 16 processing elements – configured as partitions of 1×1×16 nodes or smaller – are the next hierarchical level of parallelism, delivering up to 2.5 Tflops, double precision. Several blocks can be connected together, to build larger systems. The machine is liquid cooled, so a very large computing power can be packed in a small volume: a standard rack hosts up to 16 blocks, corresponding to a peak performance of ≈ 40 Tflops, double precision.

4. Implementation and Optimization of the LB code

In this section we describe the structure of our LB code, its data-structures, and the implementation and optimization details that we have applied to boost performance on the AuroraScience system. Due to the generality of the techniques used, our approach can be readily used for any system based on commodity Intel multi-core CPUs.

The LB approach to computational fluid-dynamics has a large degree of available parallelism, easily identified and exploited. Defining $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{c}_l \Delta t$ and rewriting the main evolution equation as:

$$f_l(\boldsymbol{y}, t + \Delta t) = f_l(\boldsymbol{y} - \boldsymbol{c}_l \Delta t, t) - \frac{\Delta t}{\tau}\left(f_l(\boldsymbol{y} - \boldsymbol{c}_l \Delta t, t) - f_l^{(eq)}\right)$$

one easily identifies the overall structure of the computation that evolves the system by one time step Δt; for each point $\boldsymbol{y}$ in the discrete grid one: i) gathers from neighboring sites the values of the fields f_l corresponding to populations that drift towards y and then ii) performs all mathematical processing needed to compute (*on a point-to-point basis*) the quantities appearing in the equation above. The key point is that both steps are completely uncorrelated for different points of the grid, so they can be parallelized according to any convenient schedule, as long as one makes sure that step i) is performed before step ii).

These features are well suited for emerging HPC architectures, that foresee a large number of processing nodes, each processing node involving many processing cores, the cores themselves often containing SIMD oriented data-paths. When porting LB algorithms for these architectures, the challenge rests in matching in the most efficient way algorithm-level parallelism with the available parallel computing resources.

Further help comes from the simple node-to-node communication pattern and timing associated to the LB method. The easiest approach is that of tiling the physical grid on the set of available processors, so that nearest-neighbor data moves translate to communications between nearest-neighbor processors assembled in a torus topology, that is becoming more and more typical in large HPC machines [8]. Finally, note that remote communications can be all concentrated at just one specific point in the main iteration loop, before population data is gathered for processing.

Each grid point has a data structure with several members. The most important is the array of 37 double-precision floating point values representing the *population*s of the D2Q37 LB code. Other members store macroscopic variables, such as velocity and temperature.

At each point in the loop over time, each grid-point is processed by several computational phases (see figure 1); we maintain two copies of the grid, each phase reading data from one copy and writing results to the other:

- stream(): this phase gathers for each site the populations according to the scheme of figure 5. This process does not perform any floating-point computation but only accesses sparse blocks of memory locations. It collects at each site all the populations that will collide at the next computational phase (collide()). For each site, this step accesses the populations of the neighbor-cells at distance 1, 2 and 3 in the physical grid.

```
typedef struct {
  double p[37];    // population array
  double u;        // velocity_x
  double v;        // velocity_y
  double rho;      // density
  double temp;     // temperature
} pop_type;

void LB-Kernel () {
  for ( step = 0; step < MAXSTEP; step++ ) {
    stream();
    bc();
    collide();
  }
}
```

Figure 1: Main data-structures and overall organization of the code. Structure pop_type collects variables associated to each grid point. Member p[] is the array of populations (see the text for details). Other members store relevant macroscopic variables.

- bc(): this phase adjusts values of the cells at the top and bottom edges of the grid to enforce appropriate boundary conditions (e.g., a constant given temperature and zero velocity).
- collide(): this phase performs all the mathematical steps needed to compute the new population values at each grid site (this is called "collision" in LB jargon). Input data for this phase are the populations gathered by the previous stream() phase. This is the floating point intensive kernel of the code; it uses only the population members of the site on which it operates, making the computation completely local.

We now address the strategies that one has to apply to exploit all the parallelism available in the LB algorithm, by mapping it at all levels available in our target architecture, namely node, core and instruction.

We start with the (relatively straightforward) steps associated to node-level parallelism. We split the grid evenly over the nodes of the system. Each node handles a portion of it, allocated on its own memory and called a *sub-grid*. A grid of size $L_x \times L_y$ is split on N_p nodes along the X dimension; we allocate on each node a strip of size $\frac{L_x}{N_p} \times L_y$. One could consider a decomposition $\frac{L_y}{N_p} \times L_x$ as well, that reduces communication requirements if $L_y \geq L_x$; however, since we plan to use our code for physics simulations in a wide range of aspect-ratios (both $L_x > L_y$ and $L_x < L_y$) and the communication overhead is small (show later for details) we arbitrarily pick up only one of the two decompositions. This allocation scheme implies an ordering of the processing nodes along a virtual ring, so each node is connected with a previous and a next node, see figure 2.

Data exchange among nodes is only needed in the stream() phase, as cells close to the right and left edges of the sub-grid of each node need data allocated on the previous and next node. We exchange all data needed for each time iteration in just one inter-node communication step, within the comm() phase. Each node sends data items belonging to the three vertical columns close to the left edge of its sub-grid to the previous node, and receives corresponding data sent from the next node. In the same way, data belonging to the right edge are sent to the next node and data belonging to the right edge of the previous node are received; this simple communication pattern is visualized in figure 2. Communication is performed at the beginning of each time step and then followed in sequence by stream(), bc() and collide(). This communication pattern – supported by a subset of the functions of our 3D-torus network – can be easily implemented on essentially any communication network. We show later that the associated overhead is small. Once this step has completed, the rest of the computation is local to the node, so all further optimizations are fully intra-node.

We now describe the sequence of intra-node optimizations that we have considered. In short, we have to make sure that i) all cores within a node work concurrently and with as little interference as possible, and ii) all floating point (vector) data-paths on each core are kept busy as much as possible. These optimizations are in principle limited by bandwidth limitations: we have to make sure that each arithmetic unit on each core timely receives from memory all data it needs: to this end we have several handles: i) reduce data traffic between memory and processor as much as made possible by the algorithmic structure (that is, re-use data already present in the processor as much as possible); ii) control memory allocation, so data is as close as possible to the core that uses it; iii) adjust the allocation strategy in order to optimize cache performance. We now provide details on these attempts, that have produced various versions of the code; performance results will be analyzed in § 5.

On each node, each phase of the computation is parallelized by splitting the sub-grid over all the cores of the node, so each one handles a different portion of it. Parallelism at core level is handled through the standard pthread library, the approach closer to the *Linux* kernel, to avoid overheads associated to e.g. openMP or MPI. To get the best performance from the node, we also run one thread per core, minimizing overheads due to the scheduling of threads among the cores. Picture 4 details the code executed on each node, which is orchestrated in the following way:

1. two threads execute the comm() phase to exchange data with neighbors. After receiving data from the neighbor node, each thread computes stream() for the three columns of its sub-grid adjacent to the sub-grid edge.
2. while step 1. executes, the remaining threads run stream() in parallel on all columns not handled by the previous threads.
3. at the end of step 1. and 2. two threads execute bc(), for the top and bottom edges of the grid respectively.
4. All threads start computing collide(), each on a different portion of the sub-grid.

Each step reads data from one copy of the grid and writes results to the other; all steps are synchronized by the use of pthread barriers.

We introduce a further level of parallelism, processing two grid sites together. Data of two cells at distance $L_y/2$ have been paired by using vector variables, see figure 3. The gcc compiler allows to define, through the vector_size extension, a new variable-type that packs two primitive types and defines vectors of two doubles. The compiler then automatically translates operations on vector variables into streaming SSE instructions.

We now focus on specific optimizations for the stream step, associated to different memory allocation options. We remind that stream() performs scattered accesses in local memory to gather populations at distance 1,2 and 3 in the grid. Performances of such step may depend on details of cache and memory-allocation policies on multi-processor Linux system, such as request-for-ownership and first-touch. We expect that the threads collaborating to this task will have limited access to shared memory locations because those threads handle different lattice sub-partitions. However we anticipate that performance will depend significantly on how well each thread is able to reuse its cached data [11, 12]. So in the following we only consider the best memory allocation choices from the point of view of cache reuse.

stream collects a subset of the populations of several neighbor grid points according to the scheme of figure 5. For each grid site, populations are allocated at successive memory locations using the ordering of their corresponding labels. This means that stream performs a sequence of sparse memory accesses. From the algorithmic point of view the labelling of populations is arbitrary, but it has an impact on performance, as it directly affects the cache hit-rate; we started with the labelling shown at left in fig. 5, and then we changed to the labelling shown at right. The latter has better cache re-use: consider e.g. populations 1,2 and 3 at $(x-2, \bar{y})$, needed when handling a grid point at some $(x, \bar{y})$ location (see again figure 5); the corresponding data items – stored in memory at contiguous addresses – are loaded on one cache line by one memory access. As we sweep the grid, moving to the next site in the y direction, $(x, \bar{y}+1)$, at least populations 2 and 3, which are needed again, are already available in cache. The gain is even larger as we consider e.g. populations $15 \cdots 21$.

A further optimization for stream() implies locking each thread to some core within one of the two processors and at the same time allocating data most used by that thread on the memory bank directly connected to that processor; this reduces the latency of memory accesses. We use the NUMA library available for Linux systems, and explicitly map the grid onto the memory bank close to the CPU of the threads accessing it. The sub-grid allocated onto each node is split in two segments, the former allocated onto memory attached to CPU0 and the latter to memory attached

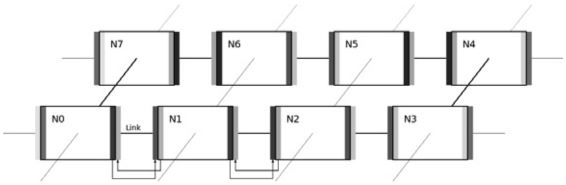

Figure 2: Virtual arrangement of the processing nodes for the program. Each node exchanges the borders of its sub-grid with the previous and the next in the ring.

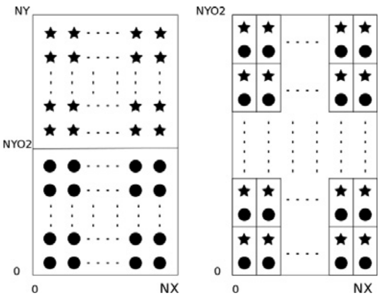

Figure 3: Cells at distance $NY/2$ are combined together, and processed by the streaming vector instructions available in the processors.

```
for ( step = 0; step < MAXSTEP; step++ ) {

  if ( tid == 0 || tid == 1 ) {
    comm();   // exchange borders
    stream(); // apply stream to left- and right-border
  } else {
    stream(); // apply stream to the inner part
  }

  pthread_barrier_wait(...);

  if ( tid == 0 )
    bc(); // apply bc() to the three upper row-cells

  if ( tid == 1 )
    bc(); // apply bc() to the three lower row-cells

  pthread_barrier_wait(...);

  collide();  // compute collide()

  pthread_barrier_wait(..);
}
```

Figure 4: Code executed by each core. The various phases are executed by a variable number of threads, and are synchronized through barriers.

to CPU1. This requires an additional copy of the edges of the two portions of the sub-grids, which is performed by a memory copy routine.

So far, our code moves data from/to memory two times, first during stream, then during collide; this can be improved upon (as suggested in [10]), as the stream and collide phases can be merged in just one computational step applied in sequence to all cells of the grid. To do this we have to take into account that computation of boundary conditions has to be done after stream but before collide. To fit this constraints, the computation steps have been structured in a different way. The computation is now organized in three main steps, each followed by a synchronization barrier:

step 1: two threads exchange the Y-borders with neighbor nodes, while two other threads perform stream for the upper and lower rows of the grid, leaving out the first and the last 3 columns; for these columns stream will be executed by threads performing communication, as soon as they receive relevant data.

step 2: two threads adjust the boundary conditions for the cells located at the top and bottom rows of the grid. After one step of synchronization, all threads (in our case 12) start computing collide for the same cells.

step 3: all threads compute stream and collide for the remaining cells of grid (the ones in the bulk).

This new schedule corresponds to another basic version of the code which we then further optimize for cache re-use and memory allocation, in the same way as described before.

All in all, we have six versions of the code: a base version that we call Ver1.1, a version including cache-related optimizations (Ver1.2), and a version also including optimized memory allocation (Ver1.3). The corresponding versions in which stream and collide are merged, are called Ver2.1, Ver2.2 and Ver2.3 respectively.

5. Performance Analysis

In this section we analyze the performance results of our simulation codes. All codes have been tested on a prototype AuroraScience machine with up to 16 nodes (192 cores). We present our analysis first for just one processing

$Lx \times Ly$	Size of data-set (GB)	Ver 1.1 (sec.)	Ver 1.2 (sec)	Ver 1.3 (sec)
252×8000	1.4	0.18	0.12 (+33%)	0.07 (+61%)
480×8000	2.8	0.35	0.25 (+28%)	0.13 (+62%)
480×16000	5.4	0.72	0.52 (+27%)	0.27 (+62%)
480×32000	11.0	1.00	0.71 (+29%)	0.54 (+46%)

Table 1: Execution time of stream for versions 1.x of the code on one node, for several sizes of the grid (and of the corresponding data set). Numbers in brackets are the improvement with respect to version 1.1.

node, we then move to a typical large scale configuration and then collect all our results in a simple performance model that exposes the scaling behavior of our implementation.

We first derive a simple estimate of the performance that we may expect on one processing node. All together, the workload W associated to mathematical processing for each grid point amounts to $\approx$ 7800 floating-point operations, consuming 37 data elements, and producing 37 updated population variables. To very first approximation, computing time T

$$T \geq \max(W/F, D/B) \tag{7}$$

where $F \approx 160$ Gflops is the peak performance of each node, D is the amount of data that must be retrieved from or written to memory, $D = 37 \times 2 \times 8$ Byte (neglecting any read-for-ownership overhead), and B is the overall memory bandwidth ($\approx$ 64 GB/sec peak, in our case). Inserting our values, we have that $T \geq \max(48.75,\ 9.25)$ ns. In other word, if peak values for performance and bandwidth apply, our application is strongly compute bound (as opposed to memory bound), so one can hope for very high efficiency. This estimate is a crude upper bound, implying that i) all vector data paths in all cores are used sustainedly, ii) data is not moved from memory to processor and vice-versa more often than needed, iii) read-for-ownership (RFO) overheads are neglected and iv) the complex addressing pattern associated to sparse memory gather operations does not degrade memory bandwidth too much (this is probably the most questionable assumption); still we should be able to reach very high efficiency for this code.

To quantify the optimization steps of the stream code, as discussed in section 4, we first run the various versions of the code on just one node with several typical grid sizes. In table 1 we show the results of this benchmarks for code versions 1.x, comparing versions 1.2 and 1.3 w.r.t. the version 1.1. For each grid size we have an improvement of about 30% with version 1.2, using a data allocation which exploits a better data cache re-use, and an improvement of about 60% if we also use the NUMA library to specify data allocation onto the two memory banks (Ver 1.3).

If we look at the column corresponding to e.g. version 1.1 (or equivalently, version 1.2) we observe a peculiar feature: as expected, the execution time of stream grows proportionally with the size of the data-base it operates upon for the smaller grids, but we observe an unexpectedly low figure when the grid size becomes very large (480×32000).

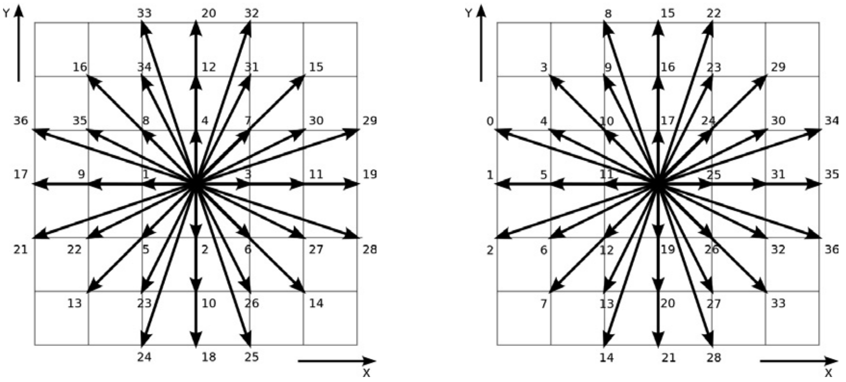

Figure 5: Visualization of the stream phase. The populations of grid points at the edges of the arrows, are gathered to the grid point at the center, at a distance 1, 2, 3 grid points. At left the original labelling of populations, at right the new labelling that, as explained in the text, improves cache performance.

dummy array size (GB)	0	3	5.5
T_{stream} (sec)	0.72	0.50	0.67

Table 2: Execution time for stream as a function of the size of the dummy array allocated by the code. 3 GB corresponds to ≈ 50% of the size of each memory bank, 5.5 GB corresponds to ≈ 90%.

	Ver. 1.1	Ver. 1.2	Ver. 1.3
T_{pbc}	0.34 s	0.25 s	0.12 s
T_{stream}	0.36 s	0.26 s	0.14 s
T_{bc}	0.9 ms	0.5 ms	0.2 ms
T_{collide}	0.39 s	0.39 s	0.39 s
$T_{\text{time/site}}$	12.5 ns	11.2 ns	8.7 ns
MLUps	78	89	115
R_{max}	23.8 %	27.0 %	35.2 %

	Ver. 2.1	Ver. 2.3	Ver. 2.3
STEP 1	0.06 s	0.06 s	0.06 s
STEP 2	1.36 ms	1.32 ms	0.64 ms
STEP 3	0.53 s	0.47 s	0.43 s
$T_{\text{time/site}}$	9.3 ns	8.7 ns	7.5 ns
MLUps	103	113	130
R_{max}	31.5 %	34.4 %	39.6 %

Table 3: Break-up of the execution time for various version of the code using 16 processing elements (192 cores). The grid size is $L_x \times L_y$ = 4032 × 16000. We also show performance (R_{max}) as a fraction of peak, and two related performance metrics: T/site (the execution time for one grid site) and its inverse (MLups, the number of grid sites updated per second).

We observe that the size of the data set in this case corresponds to almost all memory available on the node, so memory allocation is almost equally distributed among the two banks. Our guess is that in the other cases (when the size of the data set is much smaller than available memory) the operating system allocates data in an unbalanced way onto the two banks; this causes an overhead as one of the two CPUs accesses memory physically attached to the other CPU. To verify and quantify this assumption, we have modified the code, forcing the program to allocate a dummy array in memory "before" the data words actually used by the program. What we do with this trick is that we force the system to shift the allocation of data used by the program from one bank to the other; we control this shift by adjusting the size of the dummy array. Table 2 shows the execution time for stream, for a grid of size 480 × 16000 (corresponding to 5.4 GB of memory), as a function of the size of the dummy-array. Performance has a peak when the size of the dummy array is $3GB$, that is when the system is forced to allocate our data evenly on the two banks; performance is lower (and almost equal) when the size of the dummy array is 0 GB or 5.5 GB, that is when we force allocation of our data all on either of the two banks. This experiment explains the improvements of Ver. 1.3, where we apply a very fine control of the allocation of the grid variables, so the performance of stream is constant (and high) for all grid sizes. Analogous results apply also for versions 2.x of the code.

We now come to analyze performance for a typical state-of-the-art configuration, that uses 192 cores to simulate a physical system on a grid of 4032 × 16000 points. In table 3 we report the time break-up of the various routine for each version of the code that we have developed. On the left we show results for code versions 1.x and on the right for code versions 2.x.

Looking at the numbers for code versions 1.x, we find the following results:

- optimizing for cache reuse improves the computation time of the stream by ≈ 30%, and using the NUMA library has a further improvement of about 50%; this results does not apply to grid sizes using around 90% of the memory of the node, for which the performance of the version 1.1 was forced to be balanced, and the impact of using NUMA allocation is smaller.
- memory optimization steps have no impact on computation of the collide phase for which the time is strongly dominated by computation, uses data allocated to contiguous memory address, and easily exploits cache reuse.
- considering that collide also needs to read and write data from memory, we can further break up the time of collide in one part associated to memory access, of the order of 0.11 sec., assuming that full memory bandwidth can be sustained, plus a proper computational time of the order of 0.28 sec. One can therefore obtain a rough estimate of the floating-point efficiency in the inner core of the computation of 70% of the peak, that we only quote as a crude estimate of the efficiency of the floating-point engine of the processor.

For versions 2.x, in which stream and collide are merged together, we note the following:

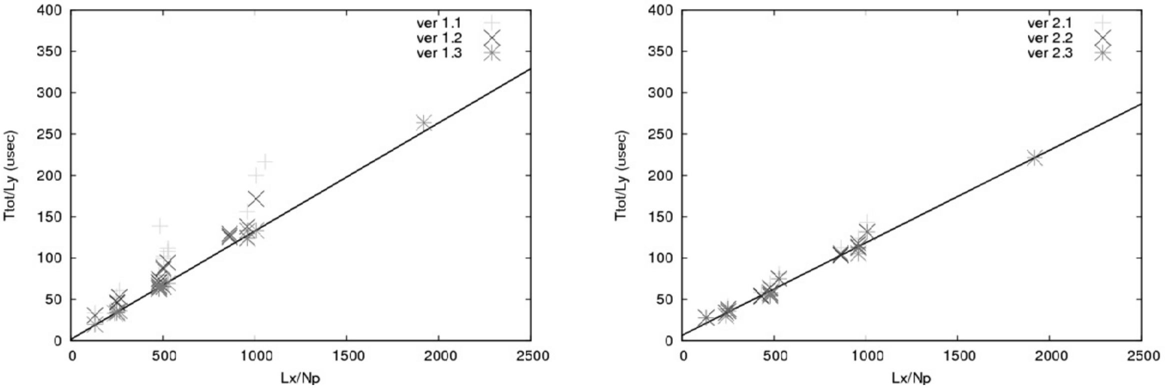

Figure 6: Plots of our measured performance data for the quantities used in equation 9, for a wide range of grid sizes and processors. The lines are fits to equation 9 for Ver1.3 (left) and Ver2.3 (right)

- as in the previous case the impact for performance of routines included in steps 1 and 2 is completely negligible
- as in the previous version memory related optimizations have an impact in step 3, which includes computation of stream and collide, and is now the only relevant routine for performance
- at global level the result performance is extremely good. In fact if we compare the stream and collide with the un-merged collide of the previous case we see that almost all the time associated to stream has been hidden in collide
- obviously the relative gain is less important because the computational pipeline is used in the same way in all cases, still we are able to increase relative performance by ≈ 20%

Globally, we reached a performance level in the range of $\approx 35 \cdots 40\%$ of peak for the various versions, for a production-grade code.

We now try to put together all our performance results for several grid sizes and numbers of used processor in a compact form, that also exhibits the overall scaling properties of our sequence of codes (this is studied in details e.g. in [13]). In principle, the complexity of our algorithm is proportional to the number of grid sites and – at the same time – the computation can be fully parallelized on any number of collaborating processors. This is true for all steps of the program, except for the comm phase, that, in our case, has to move onto each processor a data amount proportional to the L_y size of the grid. We write a very crude performance model – based on these assumptions – that estimates the wall-clock running time T_{tot} for a grid of size $L_x \times L_y$ using N_p nodes as follows:

$$T_{tot} = c_1 L_y + c_2 \frac{L_x L_y}{N_p} \tag{8}$$

where c_1 and c_2 can be fitted from measured performance data. We rewrite the previous equation as

$$T_{tot}/L_y = c_1 + c_2 \frac{L_x}{N_p} \tag{9}$$

deriving a simple linear equation that should match all our performance data.

We present our results in figure 6, where we collect our data-points for the function defined by 9, and in table 4, that shows values of the fitted parameters. Data refers to several different grid sizes, ranging from 2112×3600 to 7680×32000, and using from 4 to 16 processing nodes (48 to 192 cores). Inspection of the graphs shows that the very simple model of equation 9 correctly reproduces measured performance results. We also notice that fluctuations around the predicted value are somewhat larger for versions 1.1 and 2.1, that is in cases where no user-control on memory allocation makes performance more sensible to the automatic allocation choices made by the operating system. Looking at the table, we are able to quantify the ”asymptotic” improvements (characterized by the decreasing values of c_2) associated to the various optimization steps. We can also quantify violations to scaling, to be expected when the two terms appearing in eq. 9 become of comparable size, that is for $L_x/N_p \approx c_1/c_2 \approx 100$, a range not relevant for state-of-the-art simulations on large grids.

	Ver. 1.1	Ver. 1.2	Ver. 1.3	Ver. 2.1	Ver. 2.2	Ver. 2.3
$c_1(\mu$ sec)	21	9.8	1.5	8.4	5.3	6.4
$c_2(\mu$ sec)	0.159	0.143	0.131	0.124	0.118	0.112

Table 4: Results of the fit of our data to equation 9, for all code versions.

6. Conclusions

In this paper we have described the implementation of a multi-phase LB code in 2D and its optimization for state-of-the-art massively parallel processing systems. Our work has been tested and benchmarked on one specific HPC architectures, our results can be readily moved to most HPC system available today (or planned for the near future). Our main results are the following:

- for this class of applications/codes, inter-node parallelization is straightforward, while a carefully optimization is necessary to enable intra-node parallelism.
- relatively simple techniques are effective to obtain a good processing performance for the compute intensive part of the code, that effectively uses all the cores and data-paths available in the processors.
- memory-related optimization steps are very critical to help remove the memory bandwidth bottlenecks that arise when sparse memory accesses are necessary.
- in this direction, a cache-aware data allocation in memory is useful; even more useful is a strategy that maps data close to the processor (in our multi-CPU environment) that uses those data items most. This not surprising because the algorithm schedule forces the whole data-base to be read and written at each time step, so any cache optimization can only have a limited temporal effect.
- in conclusion, we obtain a remarkably large fraction of peak performance for a complete production code.

We plan to use this code to continue our high statistics study of convective turbulence. On the computational side, another obvious option to consider is whether GP-GPUs are an equal (or better) alternative for the problem at hand. Work is in progress in this direction.

Acknowledgments: We would like to warmly thank all members of the AuroraScience team for their efforts in bringing the AuroraScience machine on-line and making it available for our tests. The AuroraScience project is financially supported by Provincia Autonoma di Trento and Istituto Nazionale di Fisica Nucleare (Italy), and operated in collaboration with ETHlab.

References

[1] S. Succi, *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*, Oxford University Press (2001)

[2] L. Biferale et al., *High resolution numerical study of Rayleigh-Taylor turbulence using a thermal lattice Boltzmann scheme*, Physics of Fluids 22 (2010) 115112

[3] M. Sbragaglia et al., *Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria*, J. Fluid Mech. 628, (2009) 299

[4] A. Scagliarini et al., *Lattice Boltzmann Methods for thermal flows: continuum limit and applications to compressible Rayleigh-Taylor systems*, Phys. Fluids 22 (2010) 055101

[5] L. Scorzato, *AuroraScience*, Proceedings of Science, PoS(Lattice 2010)

[6] H. Baier, et al., *QPACE: Quantum Chromodynamics Parallel Computing on the Cell Boadband Engine*, Computing in Science and Engineering, 10 (2008) 46:54

[7] H. Baier, et al., *QPACE: power-efficient parallel architecture based on IBM PowerXCell 8i*, Computer Science - Research and Development 25 (2010) 149:154

[8] P. A. Boyle et al., *Overview of the QCDSP and QCDOC computers*, IBM J. Res. Dev. 49 (2005) 351:365

[9] M. Pivanti, S. F. Schifano and H. Simma, *The Torus Network Project*, Proceedings of Science, PoS(Lattice 2010)038.

[10] T. Pohl, et al., *Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes* Parallel Processing Letters, 13 4(2003) 549:560.

[11] T. Zeiser *Investigations of flow and species transport in packed beds by lattice Boltzmann simulations*, In: W. E. Nagel, W. Jäger, M. Resch (editors), High Performance Computing in Science and Engineering '06, Springer (2006) 343-354.

[12] G. Wellein, T. Zeiser, G. Hager, S. Donath *On the Single Processor Performance of Simple Lattice Boltzmann Kernels* Computers & Fluids, 35 (2006) 910:919.

[13] L. Axner, et al., *Performance evaluation of a parallel sparse lattice Boltzmann solver* Journal of Computational Physics, 227 10(2008) 4895:4911.