

# Achieving high performance in a LBM code using OpenMP

Federico Massaioli, Giorgio Amati  
CASPUR, c/o La Sapienza, P.le Aldo Moro 5, 00185 Roma, Italy  
{federico.massaioli,giorgio.amati}@caspur.it

## Abstract

This paper describes how specific architectural and implementation aspects of an SMP system (in the present work, an IBM POWER3 16-way node) influence performance and scalability of a very simple but real world code. The OpenMP parallelization of a Lattice Boltzmann Method code is presented. Scaling up to more CPUs and bigger grids successively elicits limitations arising from the hardware implementations. A sequence of substantial changes to the original implementation, aimed to overcome those obstacles, is detailed. The case study shows that an SMP system cannot be simply modelled as a bunch of processors sharing memory: making a big memory area available to each CPU in a single application can cause some processor implementation limits, unnoticed in usual serial programs, to come to evidence.

## 1 Introduction

It is a common and unfortunate pitfall that the OpenMP parallelization of a computationally intensive code is a straightforward extension of serial coding, according to the simple recipe: a) add work distribution directives; b) carefully control variable scoping; c) if needed, choose the most suitable parallel loop scheduling; d) if needed, take care of false sharing of variables among threads.

Although this approach may work with some very simple codes, most programs parallelized in this way do not scale beyond a small number of processors. It can happen that the same codes scale better when parallelized in MPI on the same SMP system, in spite of the fact that message passing causes memory to memory copies not needed with shared memory parallelization.

This paper shows how a very simple and straightforward CFD code, perfectly amenable to parallel processing, is able to elicit non trivial aspects of the CPU implementations, and requires major reworking to achieve good parallel performances in OpenMP. A deep understanding of the computing architecture and of the very CPU internals is crucial to this aim.

## 2 The Lattice Boltzmann Method

The code under scrutiny uses the Lattice Boltzmann Method (LBM)[1, 3] in the convenient BGK[2, 3] formulation to compute incompressible fluid flows in a threedimensional box. LBM is widely used for turbulence [4, 5, 6], multiphase flows [7], oil reservoir modelling[8], flows around and inside vehicles[11], and other complex flows, both in research and industrial applications.

### 2.1 Numerical scheme

The LBM numerical scheme can be clearly described as a discretization, both in coordinate and velocity spaces, of the time dependent Maxwell-Boltzmann equation for the distribution of fluid particles in phase space. As the underlying equations, it can be derived in a number of different ways. For the purpose of

the present work, it is best described as a non-PDE scheme based on a very simple kinetic model of a fluid.

The fluid is thought of as the collection of a fixed number ( $b$ ) of particle species. Particles of the same species have the same velocity vector  $\vec{c}_i$ , constant in space and time. Particle populations ( $f_i(\vec{x}, t)$ ,  $i = 1, b$ ), proportional to the number of particles of the  $i$ -th species, vary in time and space according to the flow. The macroscopic fluid quantities can be recovered from a linear superposition of those of every single species:

$$\rho(\vec{x}, t) = \sum_{i=1}^b f_i(\vec{x}, t) \quad \vec{v}(\vec{x}, t)\rho(\vec{x}, t) = \sum_{i=1}^b \vec{c}_i f_i(\vec{x}, t) \quad (1)$$

Particle populations are defined on a uniform lattice, and (in the BGK formulation) evolve in discrete time according to the master equation:

$$f_i(\vec{x} + \vec{c}_i, t + 1) - f_i(\vec{x}, t) = \omega \cdot (f_i^{eq}(\rho(\vec{x}, t), \vec{v}(\vec{x}, t)) - f_i(\vec{x}, t)) \quad (2)$$

where the parameter  $\omega$  is a relaxation parameter whose reciprocal is linear in the fluid viscosity. The local equilibrium functions

$$f_i^{eq}(\rho, \vec{v}) = d_i \rho \cdot \left( 1 + \frac{c_{i\alpha} v_\alpha}{c_s^2} + \frac{v_\alpha v_\beta}{2c_s^2} \left( \frac{c_{i\alpha} c_{i\beta}}{c_s^2} - \delta_{\alpha\beta} \right) \right), \quad \alpha, \beta \in \{x, y, z\} \quad (3)$$

( $c_s$  is the lattice speed of sound,  $d_i$  a constant depending on the lattice) come from a  $2^{nd}$  order expansion in velocity of the well known Maxwell-Boltzmann distribution.

It is evident that the velocities  $\vec{c}_i$  define the lattice structure, but their choice is not arbitrary, as strict symmetry constraints must be fulfilled for the derived quantities in eq. 1 to behave accordingly to conservation laws and Navier-Stokes equations. Moreover, the scheme is  $2^{nd}$  order accurate in space and time, in spite of its apparent  $1^{st}$  order formulation. On both issues, see [1, 3].

## 2.2 Computational aspects

As can be seen from eq. 2, time evolution results from the alternation of two physically and computationally distinct steps, collision and streaming.

The *collision* step, completely local, can be isolated as:

$$f_i(\vec{x}, t^\dagger) = f_i(\vec{x}, t) + \omega \cdot (f_i^{eq}(\rho(\vec{x}, t), \vec{v}(\vec{x}, t)) - f_i(\vec{x}, t)) \quad (4)$$

It involves only particles located at the same lattice site, and consists of a redistribution among the different species via a relaxation step toward the local state of equilibrium.

All numerical computations required by the method are performed in this step. Most of the work goes in computing  $f_i^{eq}$ , and it is to be noticed that, because of the structure of eq. 3 and of the symmetry constraints on the particle velocities, many subexpressions appear identical (or with a sign change) in the expressions for different particle species. The complete locality of this step makes it amenable to whatever form of parallelism.

The *streaming* step is non local, as particles (and consequently particle populations) “jump” from site to site, according to:

$$f_i(\vec{x} + \vec{c}_i, t + 1) = f_i(\vec{x}, t^\dagger) \quad (5)$$

Each population “moves” rigidly, synchronously, and independently, as depicted in fig. 1, no computations being performed in the process. For the vast majority of lattices and particle velocities sets, the nonlocality of this step is usually limited in range, as particles may migrate from one site to a limited number of neighbouring sites.

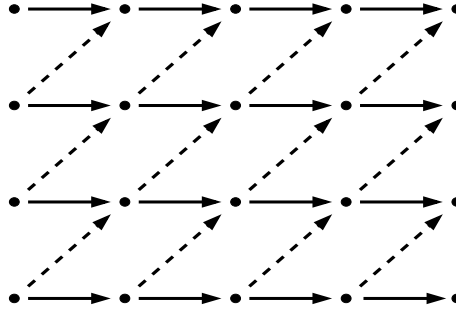


Figure 1: Two particle species with different velocities,  $(1,0)$  and  $(1,1)$ , in a 2D lattice. The streaming happens rigidly, synchronously and independently

Boundary conditions in such a kinetic model are conceptually very simple. When the lattice is periodic, particles leaving the lattice from one side, reenter at the opposite one, in the same species they belonged to. When solid walls cut the lattice orthogonally to the axes, perfectly elastic bounce-back is employed: particles never leave the lattice, but when they hit the wall their populations label is changed so that the new velocity is the opposite of the previous one (for no-slip boundaries). In both cases, this process of migration (from side to side or from species to species) is very fast, amounting to far less than 1% of the whole simulation time, and will be usually disregarded in the following. Arbitrarily shaped and/or oriented boundaries, and modelling of the correct momentum transfer at fluid/solid interfaces requires a revision of the collision process on boundaries [9], well beyond the scope of this paper.

### 3 Parallelizations goals and platform

The parallelization aimed at two results: a) a fast code suitable for collecting long time statistics of observable quantities on very long simulations (about  $10^6 - 10^7$  time steps) of turbulent 3D channel flows, on medium sized grids ( $256 \times 128 \times 128$ ); b) a foundation for a more complex code for short time simulation of two phase flows on big grids (up to  $512^3$ , more if possible). Both type of simulations requires massive data analyses, and being able to perform part of them at simulation time would greatly reduce the amount of program I/O and storage required. Some global analyses would be much faster in a shared memory, so OpenMP parallelization was chosen.

The target platform for the parallelization was the IBM “Nighthawk II” node, a 16 way SMP system based on POWER3-II processors running at 375 MHz. Each processor has a 64 KB, 512 lines, 128-way set associative L1 data cache and a direct mapped 8 MB L2 cache. The node has a theoretical peak memory bandwidth of 16 GB/s; the actual peak memory bandwidth however depends on the memory configuration, and should amount to 83% of the theoretical peak for the systems used in this work, equipped with 16 GB of RAM.

For a better understanding of next sections, it’s useful to know that in the IBM POWER architecture, the address space seen by the process is decomposed in 256 MB segments, and a translation is needed from segmented (effective) to linear virtual addresses, before the customary translation of virtual to physical memory addresses. The latter, that requires a lookup of OS kernel data structures, is supported, like in most contemporary CPUs, by a Translation Lookaside Buffer (TLB), caching relevant data for the most recently accessed memory pages. Likewise, a Segment Lookaside Buffer (SLB) fulfils the same function for the former translation from effective to virtual addresses. In the POWER3 incarnation of the architecture, the TLB is a 256 entries, 2-way set associative cache, the SLB is a 16 entries buffer.

The Guide component of the multiplatform Intel/KAI KAP/Pro ToolSet (KPTS) [10] was used for

the parallelization, in conjunction with IBM XLF 7.1 and 8.1 FORTRAN compilers. Guide transforms the code with OpenMP directives in a restructured standard FORTRAN code: parallel sections and loops are rewritten in special generated subroutines, so that a Guide proprietary runtime library, evoked by automatically inserted library calls, causes every thread to execute its part of the work. After the transformation, the thread safe native compiler is used. Guide was chosen because of its very practical and portable infrastructure for parallel profiling, and up to date tracking of the OpenMP standard. As a side benefit, OpenMP executables produced by Guide appeared consistently faster (6 ÷ 11%) than the ones produced by the native compiler. This effect is probably due to a more efficient OpenMP runtime, but further investigation (out of the scope of this paper) is needed.

## 4 Original code and basic parallelization

The FORTRAN 77 code to be parallelized was a descendant of some LBM codes (in a more complex formulation of the scheme) for vector systems. This implementation follow the traditional practice of explicitly separating the collision and streaming operations in different subroutines, so that the whole simulation can be pseudocoded as:

```
do from first time step to last time step
  call collision
  call streaming
enddo
```

The code uses a D3Q19 lattice: in standard LBM-BGK notation, this means a cubic, 3-dimensional lattice, with 19 particle species, one at rest, 18 moving<sup>1</sup>. Every population is represented with a distinct, three indexes array, with index values corresponding to lattice coordinates in units of lattice steps. Each array is accessed via a separate common block, giving the optimizer more freedom in mapping data in virtual address space to reduce TLB thrashing. The arrays are bigger then the simulated grid, as two more location per array index are needed to manage boundary conditions.

### 4.1 Basic parallelization

The collision subroutine is very easy to parallelize, due to the space locality of the process (eq. 2):

```
sub collision
  foreach gridpoint  $\vec{x}$ 
    compute  $\omega(f_i^{eq}(\vec{x}) - f_i(\vec{x}))$  for every  $i$ 
    update  $f_i(\vec{x})$ 
  endforeach
```

The outermost loop, corresponding to the rightmost array index (i.e. to the  $z$  space coordinate) can be elected for parallelization, to reduce the overhead caused by entering/exiting worksharing constructs. A `PARALLEL DO` construct, with proper privatization of variables holding intermediate results, is enough. Every iteration has the same computational cost, so that static scheduling can be used. No false sharing effect is present for non trivial grids.

The computation of the equilibrium distributions (eq. 3) is expanded in the loop body to reuse common subexpressions contributing to different species. Reading populations values from array locations to intermediate scalar variables, and a careful parenthesization of the sums corresponding to eq. 1 allows the compiler to produce better code. As a consequence, the best performing executable is produced with optimization set at -O2 level.

---

<sup>1</sup>The actual set of velocities is  $\{\vec{c}_i\} = \{(0, 0, 0), (\pm 1, 0, 0), (0, \pm 1, 0), (0, 0, \pm 1), (\pm 1, \pm 1, 0), (\pm 1, 0, \pm 1), (0, \pm 1, \pm 1)\}$ .

The nonlocality of the streaming step requires more attention. To reduce memory occupancy, the streaming step is performed in place. As the pseudocode

```

sub streaming
  foreach population  $i$ 
    foreach gridpoint  $\vec{x}$ 
      move  $f_i(\vec{x})$  to  $f_i(\vec{x} + \vec{c}_i)$ 
    endforeach
  endforeach

```

shows, different populations move independently, but care must be taken depending on the length and direction of movement, so that the destination array element can be safely overwritten. Thus, every non zero component of  $\vec{c}_i$  introduces a data dependency in the loop on the corresponding spatial index (loops are nested from greater to smaller stride, for cache efficiency reasons):

```

do k=1,n
  do j=1,m
    do i=1,1,-1                ! c_5 = (1,0,0)
      a05(i,j,k) = a05(i-1,j,k)
    enddo
  enddo
enddo

do k=n,1,-1
  do j=1,m
    do i=1,1                    ! c_18 = (0,-1,1)
      a18(i,j,k) = a18(i,j+1,k-1)
    enddo
  enddo
enddo

```

In the D3Q19 lattice, one species is at rest and is not involved in streaming. All velocities have no more than two non zero components, so that at least one loop is free of data dependencies, and the non zero components are all  $\pm 1$ . Eight particle species do not move along  $z$  (like in the first example above), so their outermost loop can be parallelized with a `PARALLEL DO`. The remaining ten populations exhibit data dependencies in the outermost loop (see second example above). Parallelization of a nested loop would be inefficient, because the OpenMP work distribution construct would be entered and exited once for every iteration of the outer loop(s), with a noticeable overhead. In the first parallelization attempt, the streaming of those ten populations was parallelized using a `PARALLEL SECTIONS` construct, one section per population.

The code generated by the compiler for the loops is reduced to the strictly required minimum at `-O2` optimization level, and no performance gains come at higher optimization levels, as tests and manual inspection of compiler assembly outputs reveal.

The resulting speedups for a 500 time steps evolution of a turbulent channel flow on a  $256 \times 128 \times 128$  grid, in double precision, (data collected via Guide) are shown in tab. 1. Please notice that the column labelled ‘total’ is not simply the sum of the other two, as it includes time for boundary conditions, I/O reports, and diagnostic checks. The streaming step is a very poor performer: only a speedup of 6 using 16 processors is reached.

The approach presents an obvious and serious limitation: the cost of streaming is practically the same for every populations, and the `PARALLEL SECTIONS` construct is unbalanced when the number of processors is not an even divisor of 10, as can be seen (fig. 2) using GuideView [10] on profiling data collected by Guide runtime.

| # procs | total        | collision    | streaming   |
|---------|--------------|--------------|-------------|
| 1       | 1882" (1.00) | 1248" (1.00) | 573" (1.00) |
| 2       | 957" (1.97)  | 628" (1.99)  | 291" (1.97) |
| 4       | 519" (3.63)  | 321" (3.89)  | 171" (3.35) |
| 8       | 323" (5.83)  | 174" (7.17)  | 125" (4.58) |
| 16      | 227" (8.29)  | 110" (11.4)  | 96" (5.97)  |

Table 1: Timing in seconds (speedup) for basic parallelization ( $256 \times 128 \times 128$ , double precision, 500 time steps)

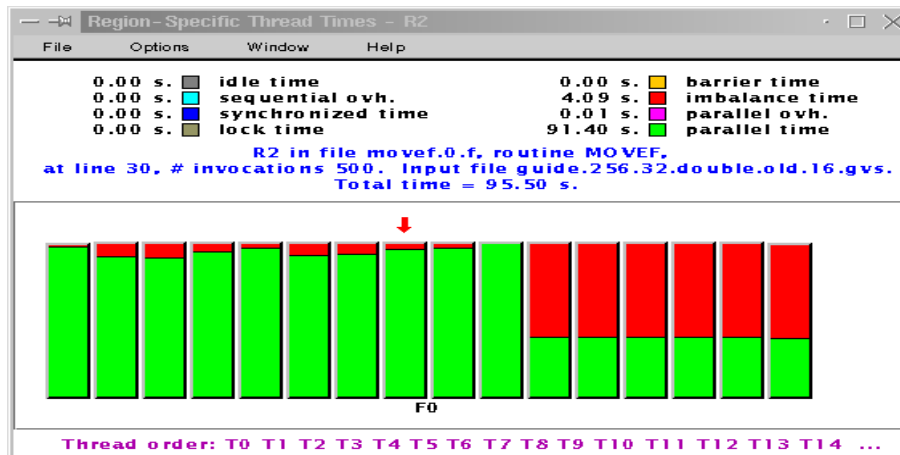


Figure 2: Unbalance of streaming step using 16 threads

| # procs | total        | collision    | streaming   |
|---------|--------------|--------------|-------------|
| 1       | 1928" (1.00) | 1253" (1.00) | 614" (1.00) |
| 2       | 980" (1.97)  | 629" (1.99)  | 312" (1.97) |
| 4       | 541" (3.56)  | 329" (3.81)  | 178" (3.45) |
| 8       | 294" (6.56)  | 168" (7.46)  | 103" (5.96) |
| 16      | 211" (9.14)  | 102" (12.3)  | 87" (7.06)  |

Table 2: Timing in seconds (speedup), for balanced basic parallelization ( $256 \times 128 \times 128$ , double precision, 500 time steps)

## 4.2 Balanced basic parallelization

In a brute force attempt to remove this unbalance loops could be grouped in  $m$  separate sections of a `PARALLEL SECTIONS`, where  $m$  is a multiple of the number of threads. Unfortunately, in this approach one has to modify the code each time according to the number of available threads, and a perfect matching of threads to `sections` is again not generally possible<sup>2</sup>.

The unbalance problem can be overcome thanks to the inner working of Guide (see sec. 3). It is possible to restructure the 10 streaming loop nests for populations moving along the  $z$  axis with a simple loop exchange, so that the outermost loop is free of data dependencies, as in:

```
do i=1,1 ! c_18 = (0,-1,1)
  do k=n,1,-1
    do j=1,m
      a18(i,j,k) = a18(i,j+1,k-1)
    enddo
  enddo
enddo
```

The outermost loop is now parallelizable, and no overhead due to an iterated `PARALLEL DO` is present. The loops appear ordered in a cache inefficient way, but Guide rewrites the loop as a serial, parameterized subroutine to be called by its runtime, so the compiler can rearrange the loop nest for optimal cache performance. An inspection of the annotated assembly listing produced by XLF confirms that. As even a barely decent optimizing compiler would suffice for the job, this technique should be generally applicable to other systems and codes.

The results of this approach, again for a 500 time steps evolution of a  $256 \times 128 \times 128$  grid, in double precision, are shown in tab. 2. It must be noted that the timings for the collision step changes from tab. 1 to tab. 2, even if no modifications were done to the corresponding subroutine. This can happen, as patterns of access of CPUs to data changes from collision to streaming, and those in the latter were affected by the modifications bringing to tab. 2 results. Moving from this test case (608 MB of data) to a smaller one fitting the total amount of L2 caches ( $N_{CPU} \times 32MB$ ), would make the effect much more apparent.

The new approach doesn't produce a significantly better speedup. This shouldn't be surprising: a look at the histogram in fig. 2 shows that the unbalance cannot account for more than 25% of loss of parallel efficiency, while data in tab. 1 reveal a loss of more than 60%.

The real bottleneck is the memory bandwidth request. As reported above, the streaming loop nests are reduced by the compiler to the very minimum code required to move data from memory to memory according to the ordering specified in the loop bounds and increments. In double precision, 576 MB

<sup>2</sup>It is a well known fact that, as the number of CPUs in an SMP system is usually a power of 2, users tend to allocate them in power of 2 as well.

| # procs | total        | collision    | streaming   |
|---------|--------------|--------------|-------------|
| 1       | 1461" (1.00) | 1074" (1.00) | 347" (1.00) |
| 2       | 751" (1.95)  | 540" (1.99)  | 183" (1.87) |
| 4       | 384" (3.80)  | 271" (3.96)  | 94" (3.69)  |
| 8       | 209" (7.00)  | 138" (7.78)  | 55" (6.31)  |
| 16      | 130" (11.2)  | 74" (14.5)   | 42" (8.62)  |

Table 3: Timing in seconds (speedup), for balanced basic parallelization using single precision ( $256 \times 128 \times 128$ , 500 time steps)

of memory must be read and then written at every time step, for a total of 1.125 GB per time step. The speedup obtained are compatible with a sustained memory bandwidth (for this specific application) around 6.5 GB/s, well below the maximum theoretical bandwidth allowed by the system. Sustainable memory bandwidth appears saturated using more than 8 processors. A comparison with a run of the same test case in single precision (tab. 3) confirms this picture: the time needed for the streaming step is practically halved, as is halved the amount of data to move back and forth between CPUs and memory.

## 5 Fused implementation

According to results in previous section, OpenMP parallelization of the classical LBM implementation is unsuccessful, at least when using more than 8 processors and double precision. To reach better performance the implementation must be rethought and the original code widely modified.

Comparing data in tab. 2 to those in tab. 3, it is apparent that, switching from double to single precision, the reduction of times for collision is less than that for streaming. The collision step as well reads all the populations and writes them back, and it does even more memory accesses than streaming does, as the latter disregards the population at rest. In the collision, however, all values are modified through a number of floating point operations (a total of  $\sim 260$  per loop iteration), whose cost is the same for single and double precision. This suggests that the time needed for calculations makes data accesses significantly less relevant, partly concealing memory access costs.

The collision step would thus be the natural place to hide the cost of memory access operations, implementing the master equation (2) in a single fused step. This can not be done in the present implementation. Writing the new populations values coming out of the collision in lattice sites different from the one they were read from, would halve the total number of memory accesses, at the price of unsurmountable data dependencies in all the three nested loops on spatial indexes. Using two copies of the populations arrays, alternately reading from one copy and writing to the other one, would solve the problem at the expense of doubling the amount of memory used, thus making big grids unaffordable.

The solution can be found getting back to the kinetic view of the fluid. All the particles of a same species move rigidly at every time step by a fixed amount. In the implementation described above, each population is represented in a Eulerian approach, in the frame of reference where the lattice is at rest. In computational terms, for every array representing a different particle species population, the mapping of array indexes to physical space coordinates is fixed, so that the values must be migrated at each time step, according to the corresponding velocity. In a Lagrangian approach, each populations is represented in the frame of reference in which the corresponding particles are at rest. In computational terms, for every array representing the population of a particle species, the values are kept (stored) at rest, while the mapping of array indexes to physical space coordinates changes with time (see fig. 3). The needed transformation maps periodically the spatial coordinates on loop indexes, and change cyclically in time, according to the corresponding velocity components. It can be simply realized with linear congruences:

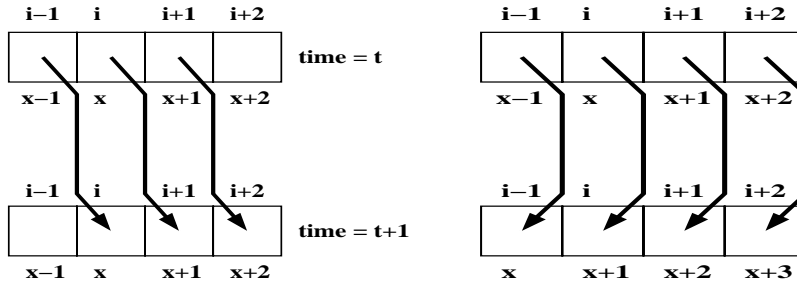


Figure 3: A 1-D pictorial view of the basic (left) and fused (right) implementation of streaming, showing how for the latter data stay in the same locations while mapping of physical coordinates to array indexes is changed.

| # procs | total        | collision    |
|---------|--------------|--------------|
| 1       | 1365" (1.00) | 1345" (1.00) |
| 2       | 706" (1.93)  | 688" (1.95)  |
| 4       | 374" (3.65)  | 357" (3.77)  |
| 8       | 206" (6.63)  | 189" (7.12)  |
| 16      | 131" (10.4)  | 115" (11.7)  |

Table 4: Timing in seconds (speedup), for fused parallelization ( $256 \times 128 \times 128$ , double precision, 500 time steps)

$index_\alpha(x_\alpha, c_{i\alpha}, t) = (x_\alpha - c_{i\alpha} \cdot t) \bmod L_\alpha$ , where  $L_\alpha$  are the grid side lengths.

With this approach, collision is still completely local, but the populations entering the collision process for each physical lattice site come from different places, according to their present mapping of physical coordinates to array indexes. Nevertheless, arrays are accessed in order, at unitary stride. One added benefit of this implementation is that periodic boundary conditions come at no cost.

The resulting timings and speedups are reported in tab. 4 (again for 500 time steps evolution of a  $256 \times 128 \times 128$  grid, in double precision). It should be noticed that while the speedup is not that higher, this is an effect of the remarkable improvement in code speed, even on a single CPU, so remarkable that one could think that the new code can be applied to very big grids, the second target of the parallelization.

Unfortunately, in a test on a  $512 \times 256 \times 256$  grid, in double precision, the program appears more than twice slower, at  $1.7 \times 10^{-6}s$  per grid point, with respect to timings on smaller grids:  $6.7 \times 10^{-7}s$  per grid point in double precision,  $5.6 \times 10^{-7}s$  in single. A more careful investigation shows that the problem has to do with the process data size, as the computing time grows when the data size approaches 4 GB, and stays constant from 4.5 GB up (see tab. 5).

The fused implementation is still better than the basic one at those grid size. In fact, on a  $512 \times 256 \times 256$  grid, the fused collision subroutine takes 1157s for 20 time steps, while the basic collision subroutine takes 1100s. Nevertheless, keeping into account the streaming subroutine, absent in the fused implementation, the basic one is still slower, requiring a total of 1291 for 20 time steps. An MPI version of the program, based on a simple domain decomposition and using the fused implementation on each subdomain, behaves normally. The MPI version is not affected by the slowdown as each of the  $n$  processes holds  $1/n$  of the total data, so the process size is naturally bounded.

| Grid size                   | Data size | seconds/gridpoint     |
|-----------------------------|-----------|-----------------------|
| $256 \times 128 \times 128$ | 608 MB    | $0.68 \times 10^{-6}$ |
| $460 \times 230 \times 230$ | 3527 MB   | $0.78 \times 10^{-6}$ |
| $480 \times 240 \times 240$ | 4007 MB   | $1.23 \times 10^{-6}$ |
| $500 \times 250 \times 250$ | 4529 MB   | $1.66 \times 10^{-6}$ |
| $512 \times 256 \times 256$ | 4864 MB   | $1.67 \times 10^{-6}$ |
| $690 \times 345 \times 345$ | 11905 MB  | $1.66 \times 10^{-6}$ |

Table 5: Time per gridpoint as a function of grid size, for fused implementation in double precision

| Size                                   | Tot SLB     | Tot TLB   | SLB/site | TLB/site |
|--|-------------|-----------|----------|----------|
| $256 \times 128 \times 128$ (0.59 GB)  | 522177      | 9813432   | 0.006    | 0.117    |
| $460 \times 230 \times 230$ (3.44 GB)  | 80544498    | 87991790  | 0.165    | 0.181    |
| $480 \times 240 \times 240$ (3.91 GB)  | 2977350993  | 128279951 | 5.618    | 0.242    |
| $500 \times 250 \times 250$ (4.42 GB)  | 5893470269  | 96659516  | 9.430    | 0.155    |
| $506 \times 253 \times 253$ (4.58 GB)  | 6258325109  | 287137680 | 9.661    | 0.443    |
| $512 \times 256 \times 256$ (4.75 GB)  | 6770726198  | 690491384 | 8.781    | 1.029    |
| $518 \times 259 \times 259$ (4.92 GB)  | 7332936007  | 67871566  | 10.551   | 0.098    |
| $690 \times 345 \times 345$ (11.6 GB)  | 27792822975 | 364585734 | 16.921   | 0.222    |
| $512 \times 256 \times 256$ (4.75 GB)* | 7744183249  | 191944933 | 11.540   | 0.286    |

Table 6: SLB and TLB misses, total and per lattice site, for the fused implementation in double precision. The last line reports a test for the basic implementation, for the grid size corresponding to the peak in TLB misses for the fused one.

## 6 Bundled implementation

The cause of this performance degradation lies in the internals of the POWER3 CPU. Tab. 6 reports a few measurements collected with CPU internal hardware counters via some tools available from IBM (hpmcount and HPMlib [12]). Fused implementation appears to stress the TLB and SLB processor structures, depending on the grid size.

The POWER3 SLB has 16 entries, so that, to benefit from fast effective-to-virtual address translation, the total number of segments accessed in a single loop iteration must not span more than 4 GB ( $16 \times 256$  MB). Both basic and fused implementation access 19 different memory locations at every iteration of the collision loop nest. Those 19 memory locations are in different arrays, each one with slightly more elements than the lattice points (because of the additional space needed for boundary conditions). As the lattice size grows, those location will be farther apart, and they will eventually fall in different memory segments, for big enough grids, thus exceeding the capabilities of the SLB.

The TLB abuse grows as well as a function of the grid size, but for different reasons. In fact, as 19 memory pages are accessed for every lattice site, and once a 4 KB page has been accessed, the related entry in the TLB should be used for a total of 512 lattice sites in double precision (the lattice is walked through at unitary stride), the 512 entries TLB should not suffer much because of it. However, as the POWER3 TLB is 2-way set associative, it is subject to thrashing: pages whose virtual addresses are a multiple of 512 KB apart contend for the same two entries. The compiler, by adjusting the relative position in address space of populations arrays, is able to reduce TLB thrashing for the basic implementation, where the distances among the 19 memory addresses accessed for every site collision is fixed along the whole run. In the fused implementation, however, the distances among those 19 addresses changes at every

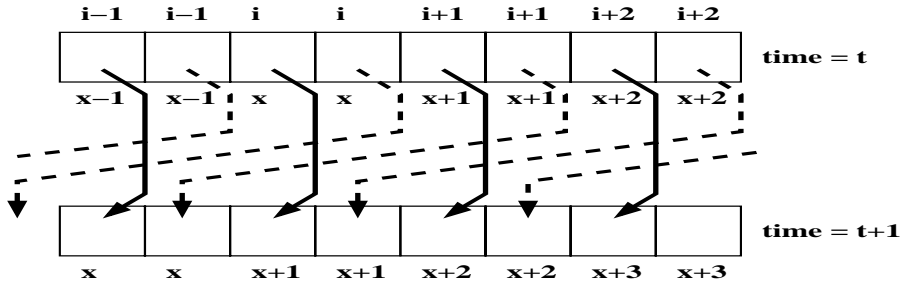


Figure 4: A 1-D pictorial view of the relative streaming of one of the particle species in a bundle, with respect to the other.

time steps, as the mappings of physical coordinates to array indexes vary with time for every population, independently from the others. As a consequence, a cyclic pattern is generated<sup>3</sup>, in which the amount of TLB thrashing per time step oscillates continuously between an optimal minimum and a performance destructive maximum. A comparison between fused and basic implementation (last line of tab. 6) for a  $512 \times 256 \times 256$  grid evidences this effect<sup>4</sup>.

The only way to reduce SLB abuse is to reduce the number of memory streams in input and output of the collision loop. TLB thrashing should benefit as well from this strategy, as less memory streams means less memory pages accessed for every iteration, and consequently less contenders for TLB entries, even in the worst case scenario. Measurements with a different code (an OpenMP pseudospectral Navier-Stokes solver[13]), using far less memory streams and showing no degradation as the grid size grows, support this view. The reduction should possibly preserve the benefits of the fused implementation.

Once again, the kinetic view helps to find the solution. Let's take two particle species  $i$  and  $j$  whose corresponding velocities differ in only one component,  $\alpha$ . In the frame of reference in which the first species is at rest, particles of the second one move at every time step by a fixed, usually small number of lattice sites along the  $\alpha$  axis. If the corresponding populations are bundled together in a single four indexes array (the leftmost index identifying the two species, the remaining ones corresponding to the coordinates in the frame of reference in which the first species is at rest) every time that a collision in a site is performed, the new value of  $f_i$  must be written back to the same place, the one for  $f_j$  must be written back with a displacement of  $c_{j\alpha} - c_{i\alpha}$  along the  $\alpha$  axis (see fig. 4). This sort of “relative streaming” of the second species with respect to the first one can be done in a cache-, SLB-, and TLB-friendly way.

The most convenient way to proceed for a D3Q19 lattice is to bundle together populations with the differing velocity component such that  $c_{i\alpha} = +1$  and  $c_{j\alpha} = -1$ . For  $\alpha = x$ , 5 bundles are possible: the post collision value of  $f_j$  must be written 2 sites back (overwriting a location not needed anymore, because of the loop order), i.e. 32 bytes back for double precision. 75% of the times, this location will fall in the same L1 cache line. Moreover, almost always it will be in an already cached line and in the same memory page. For  $\alpha = y$ , 3 more bundles are possible. For similar arguments, the post collision value of  $f_j$  will be written back in a different L1 cache line, but most of the times the line will be already cached. The memory page will be different as well, but a contention for the same associativity set in the TLB is improbable, as it would require the grid  $x$  size to be a multiple of 16384. No bundling for  $\alpha = z$

<sup>3</sup>Analytic, per timestep measurements supporting this explanation cannot be provided, as the measurement causes a failure in HPMLib, unsupported alpha software kindly provided by IBM at no cost.

<sup>4</sup>A synthetic benchmark, artificially causing a SLB miss or a TLB miss at every loop iteration of a simple sum, shows that the former is able to cause a 25% slowdown and that the latter costs 3.1 times more. Nevertheless, the combined impact of both effects on a real code is of course determined by the actual mix of the two.

| Size                                  | Tot SLB  | Tot TLB   | SLB/site | TLB/site |
|---------------------------------------|----------|-----------|----------|----------|
| $256 \times 128 \times 128$ (0.59 GB) | 0        | 20730167  | 0.000    | 0.247    |
| $460 \times 230 \times 230$ (3.44 GB) | 7197420  | 55250725  | 0.015    | 0.113    |
| $480 \times 240 \times 240$ (3.91 GB) | 11735480 | 63816890  | 0.021    | 0.115    |
| $500 \times 250 \times 250$ (4.42 GB) | 13395834 | 74581247  | 0.021    | 0.119    |
| $506 \times 253 \times 253$ (4.58 GB) | 13371364 | 70764631  | 0.021    | 0.109    |
| $512 \times 256 \times 256$ (4.75 GB) | 14449740 | 242264763 | 0.021    | 0.361    |
| $518 \times 259 \times 259$ (4.92 GB) | 14933566 | 70260153  | 0.021    | 0.101    |
| $690 \times 345 \times 345$ (11.6 GB) | 35367086 | 186552782 | 0.021    | 0.114    |

Table 7: SLB and TLB misses, total and per lattice site, for the bundled implementation in double precision.

is feasible, as it would introduce data dependencies on the outermost loop, inhibiting its parallelization. Moreover, some related modifications must be applied to the boundary condition routine.

The new bundled implementation access 11 memory streams in the collision loop, significantly relieving SLB and TLB activities, as the results in tab. 7 confirm. The time per gridpoint of the new implementation is  $7.0 \times 10^{-7}$  s, and no significant degradations were observed for the grid sizes allowed by the RAM in the system.

## 7 Preliminary results on POWER4

The POWER3 is not anymore the leading CPU in the IBM POWER architecture, as the POWER4 already entered the market. Preliminary tests demonstrated that the degradation of the fused implementation timings on POWER3 processors, for grids occupying 4 GB and more, is not observed on the POWER4. This is consistent with the fact that in the POWER4 implementation, SLB is a 64 entries, fully associative cache, and the TLB has 1024 entries, 4 way set associative. Additionally, POWER4 supports large, 16 MB memory pages, to further reduce impact of adverse memory access patterns on the TLB. In view of this, was the bundled implementation, aimed at solving the problem on the POWER3, worth the effort?

The general validity of the fused and bundled implementation approaches could be questioned. Adverse memory access patterns could arise on Distributed Shared Memory (DSM) systems, as while the mappings of physical coordinates to array indexes changes in time, a scenario could happen in which most of the data accessed by every CPU resides in remote memory banks. It is not clear whether the reduction in memory accesses resulting from the fused approach outweighs such effects.

In some sense, POWER4 is a DSM architecture: the pSeries 690 Model 681 system has one memory bank every 4 CPUs. The access to remote memory banks goes, however, through the L3 caches, which are effectively shared by all the CPUs in the node, via the interconnection network. L2 caches as well are shared by the two processors on the same chip, and participate in the memory coherence enforcement. Such a novel architecture is yet to be understood in all its performance implications.

On the other side, the bundled implementation could be beneficial for the POWER4 as well, because it reduces memory streams and lessens contention on TLB entries. In fact, while the latter could be reduced using large memory pages, this approach is not always valid for multithreaded applications, as it raises the probability of memory accesses to remote memory banks, and the present system management of large pages poses significant constraints on the system adaptability to mutating workloads. Finally, the hardware prefetch feature of POWER4 is optimized for 4 to 8 memory streams at the same time, and the 19 ones needed by the basic and fused implementation are definitely too much.

| # procs | Case A | fused  | Case A | bundled | Case B | fused   | Case B | bundled |
|---------|--------|--------|--------|---------|--------|---------|--------|---------|
| 1       | 780    | (1)    | 592    | (1)     | 22772  | (1)     | 16350  | (1)     |
| 2       | 412    | (1.89) | 322    | (1.84)  | 11463  | (1.99)  | 8443   | (1.94)  |
| 4       | 202    | (3.86) | 168    | (3.52)  | 5579   | (4.08)  | 4371   | (3.74)  |
| 8       | 105    | (7.43) | 97     | (6.10)  | 3111   | (7.32)  | 2337   | (7.00)  |
| 16      | 55     | (14.2) | 58     | (10.2)  | 1685   | (13.5)  | 1352   | (12.1)  |
| 32      | 32     | (24.4) | 44     | (13.5)  | 975    | (23.36) | 939    | (17.4)  |

Table 8: Timing in seconds (speedup) on a POWER4 pSeries 690 node, for fused and bundled parallelization, double precision, 500 time steps. Case A corresponds to a  $256 \times 128 \times 128$  grid (0.6 GB of data), case B to a  $768 \times 384 \times 384$  grid (16 GB of data).

The preliminary results shown in tab. 8 were obtained on a 32 CPUs pSeries 690 system using small memory pages. The bundled implementation has a definite advantage, which however decreases as the number of processors increases, at least for smaller grids. More detailed measurements and analyses are anyhow needed to fully understand all the issues. The poor scaling exhibited at 32 CPUs is a known problem, that could be caused by memory bandwidth limitations on the fully loaded system (running at 1.3 GHz, the POWER4 is much more dependent on fast memory access) or by the AIX scheduler behaviour, and needs further investigation.

## 8 Conclusions and future work

From this work, some general conclusions can be drawn. Even a very simple and regular code, an apparently ideal candidate for a straightforward OpenMP parallelization, can exhibit interesting behaviour and calls for deep understanding of the algorithm and the implementation.

Scaling to higher number of CPUs can bring the program to stress the system in unforeseen ways. In the present case, extensive and worthwhile reworking were required, with the serial and MPI version benefiting as well from the improvements. It must be stressed that some of the problems discussed in this work, where not apparent until the speed and scalability of the application made simulations on big grids feasible, and cannot appear when the process data size is small.

Profiling data for a multithreaded application must be taken with more care than those for a serial one, as they tend to emphasize the relative performance of different threads, while a general slowdown could be hidden. A theoretical and predictive model for the performance of SMP systems, still lacking, would be of great help in detecting such issues. However, the number of implementation details that, like those described in this paper, can affect performances of multithreaded applications is so high, and their effect so difficult to predict, that building such a model is by no means an easy task.

The often heard statement that MPI parallelization of a program is more difficult with respect to OpenMP is not correct, as the total effort needed to scale to higher number of processors is probably the same. In the present case, the MPI version was free of some problems, because the single MPI process is small enough to avoid them. The numerical scheme described exhibits good parallel properties, and it should scale as well in MPI than in OpenMP, with a small advantage for the latter, which avoids memory to memory communications inside the SMP node. After reworking the implementation, the OpenMP version behaves slightly better than the MPI one, as it should be, the cause of the poor scaling of OpenMP being the result of some implementation details of a system component.

It is to be expected that, as SMP systems become bigger and more powerful, more and more cases as the present one will have to be faced. A deep understanding not only of the system architecture, but of the CPU internal details as well, are needed to explain and overcome performance problems. As MPI

programmers learn, a powerful cluster is not simply the connection of a number of powerful systems via a blazingly fast network. Likewise, a performing SMP system is not built just from powerful CPUs and a blazingly fast memory subsystem: bigger SMPs allow for bigger computations and those can expose components limitations and unforeseen interactions.

The new LBM implementations reported in this paper require additional investigations. Not all architectures adopt a segmented memory model, but TLBs are standard parts of every CPU, so the bundled implementation approach could prove effective on other hardware platforms. Some of the modifications discussed in this paper, while of help for reducing abuse of the memory subsystem, improving cache exploitation, and taking advantage of data prefetching, could cause performance degradation on DSM systems. Apparently, many vendor (IBM with Power4 and successors, HP/COMPAQ with Alpha EV7) are introducing flavours of DSM, albeit tightly integrated with the CPU. Both issues must be experimentally investigated.

Last but not least, the implementation changes discussed in this paper are effective for regular grids on very regular domain. Some preliminary ideas about bringing those advantages to different fields of application of LBM, requiring more complex domains and grids, must be further developed.

## Acknowledgements

We warmly thanks Massimo Bernaschi (CNR-IAC) for useful discussions and deep insights in the IBM Power architecture. Giorgio Richelli (IBM EMEA) gave us detailed information about Power3 internals. CINECA gave us access to a not yet publicly available IBM Power4 installation, our first tests on it were helpfully supported by Giovanni Erbacci and Cristiano Calonaci. IBM EMEA gave us access to a POWER4 node for a more extensive testing of the fused and bundled implementations on the new architecture. Last but not least, we thanks Andrei Maslennikov and Giuseppe Palumbo (CASPUR systems group) for timely support on our IBM Power3 SP system.

## References

- [1] R. Benzi, S. Succi, M. Vergassola, *Theory and Application of the Lattice Boltzmann Equation*, Phys. Reports, 222 (3), 147, (1992)
- [2] Y. H. Qian, S. Succi, S. Orszag, *Recent Advances in Lattice Boltzmann Computing*, Ann. Rev. Comp. Phys., 3, 195, (1995)
- [3] S. Succi, *The Lattice Boltzmann Equation - For Fluid Dynamics and Beyond*, Oxford University Press, (2001)
- [4] F. Massaioli, R. Benzi, S. Succi, *Exponential Tails in Two-Dimensional Rayleigh-Bénard Convection*, Europhys. Lett., 21 (3), 305-310, (1993)
- [5] F. Massaioli, R. Tripiccion, et al., *LBE Simulations of Rayleigh-Bénard Convection on the APE100 parallel processor*, Int. J. Mod. Phys. C, 4 5, 993-1006, (1993)
- [6] R. Benzi, G. Amati, C. Casciola, F. Toschi, and R. Piva, *Intermittency and scaling laws for wall bounded turbulence*, Phys. of Fluids, Vol. 11, No. 6, pp. 1 – 3, (1999)
- [7] E. Orlandini, M.R. Swift, and J.M. Yeomans, *A Lattice Boltzmann Model of Binary Fluid Mixture*, Europhys. Lett., Vol. 32, No. 6, pp. 463 – 468, (1995)
- [8] R. D. Hazlett, S. Y. Chen, W. E. Soll, *Wettability and rate effects on immiscible displacement: Lattice Boltzmann simulation in microtomographic images of reservoir rocks*, J. of Petroleum Sc. & Eng., Vol. 20, 3-4, pp. 167 – 175,(1998)

- [9] H. Chen, K. Molvig, C. Teixeira, *Realization of Fluid Boundary Conditions Via Discrete Boltzmann Dynamics*, Int. J. of Modern Physics C, Vol. 32, p. 1281, (1998)
- [10] [www.kai.com](http://www.kai.com)
- [11] [www.exa.com](http://www.exa.com)
- [12] [www.alphaworks.ibm.com](http://www.alphaworks.ibm.com)
- [13] F. Bonaccorso, F. Massaioli, *An OpenMP Pseudospectral Navier-Stokes Solver for IBM Power3 SMP Systems*, in preparation