Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 1004–1013

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Lattice Boltzmann Simulation Code Optimization Based on Constant-time Circular Array Shifting

G. Dethier[a,*], P. A. de Marneffe[a], P. Marchot[b]

[a]*Department of Electrical Engineering & Computer Science of University of Liège*
[b]*Department of Chemical Engineering of University of Liège*

Abstract

Lattice Boltzmann (LB) methods are a class of Computational Fluid Dynamics (CFD) methods for fluid flow simulation. LB simulation codes have high requirements regarding memory and computational power: they may involve the update of several millions of floating point values thousands of times and therefore require several gigabytes of available memory and run for several days. Optimized implementations of LB methods minimize these requirements.

An existing method based on a particular data layout and an associated implementation implying a constant time array shifting allows to reduce the execution time of LB simulations and almost minimize memory usage when compared to a naive implementation.

In this paper, we show that this method can be further improved, both in memory usage and performances by slightly modifying the data layout and by using blocking in order to enhance data locality.

Keywords: Lattice Boltzmann, Optimization, Data Locality, Circular arrays

1. Introduction

Lattice Boltzmann (LB) methods [1, 2] are a class of Computational Fluid Dynamics (CFD) methods for fluid flow simulation. Unlike many CFD methods solving motion equations at some points in space, LB methods use an alternative approach: fluid is described by fictitious particles moving on a regular grid. These particles collide with each other or against solid obstacles at the cells of the mesh. Flow parameters are then computed in function of the state of these particles.

LB methods are particularly interesting for simulating fluid flows in complex boundaries like porous media. However, LB simulation codes have high requirements regarding memory and computational power: they may involve the update of several millions of floating point values thousands of times and therefore require several gigabytes of available memory and run for several days. Optimized implementations of LB methods minimize these needs.

LB simulation codes imply the regular access to values in large multi-dimensional arrays stored in main memory and the update of these values. An obvious way to optimize a LB simulation code is to ensure data locality, in particular by choosing an optimal way to organize data in memory [3, 4, 5, 6].

[*]Corresponding author
Email address: `G.Dethier@ulg.ac.be` (G. Dethier)

1877–0509 © 2011 Published by Elsevier Ltd. Selection and/or peer-review
under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.105

1877–0509 © 2011 Published by Elsevier Ltd. Selection and/or peer-review
under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.106

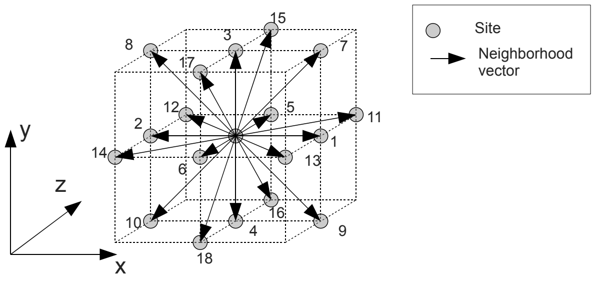

Figure 1: A site (center of the cube) of a D3Q19 lattice and its neighborhood. The numbers are identifiers of the neighborhood vectors (i is the identifier of vector $\mathbf{n}_i$).

Murphy [7] proposes an interesting data layout and an associated implementation that allows to reduce the execution time of LB simulations. However, Murphy's method is not adapted to programming languages that do not support pointer arithmetic. Also, it can be enhanced in order to further reduce execution time. Finally, it implies a small memory overhead that can be avoided. In this paper, we solve all these drawbacks.

All the algorithms described in the following are written using the guarded commands language defined by Dijkstra [8].

Section 2 shortly describes LB methods and a naive implementation. In Section 3.1, the data locality principle is presented and the representation of multi-dimensional arrays is discussed. Section 3 describes the method proposed by Murphy and a first improvement. Section 4 describes how Murphy's method can be further enhanced by ensuring better data locality. Execution times obtained using the naive implementation of Section 2, Murphy's method and our optimized implementation are compared in Section 5. Finally, Section 6 concludes this paper.

2. Lattice Boltzmann Methods

In LB methods [1, 2], the spatial domain ($\mathbb{R}^3$) is discretized into a regular grid called *lattice*. Each node of the lattice is called a *site* and is associated to a point in space. If the point associated to a site is part of a solid, the site is an *obstacle*. Time is also regularly discretized. Finally, the set of possible velocities for a particle ($\mathbb{R}^3$) is reduced to a set of q velocity vectors $\mathbf{v}_i \in \mathbb{R}^3$ with $i = 0, 1, \ldots, q-1$. These vectors are defined such as a particle at point $\mathbf{p}$ in space at time t and moving according to a velocity $\mathbf{v}_i$ is at point $\mathbf{p} + \mathbf{v}_i$ at time $t + \Delta t$ where Δt is the time sampling period.

A site has a *position* $\mathbf{x}$ in the lattice with $\mathbf{x} \in \mathbb{Z}^3$. A function $s : \mathbb{Z}^3 \to \{true, false\}$ is used to define the nature of a site: if $s(\mathbf{x})$ is *true* then the site at position $\mathbf{x}$ is an obstacle. Otherwise, the site is not an obstacle.

A set of q *neighborhood vectors* $\mathbf{n}_i$ with $i = 0, 1, \ldots, q-1$ and $\mathbf{n}_i \in \mathbb{Z}^3$ is defined such as $\mathbf{x} + \mathbf{n}_i$ is the position of another site of the lattice. $\mathbf{n}_i$ has the same direction as $\mathbf{v}_i$ and is defined such as a particle at the point associated to position $\mathbf{x}$ at discrete time t and moving according to velocity $\mathbf{v}_i$ is at the point associated to position $\mathbf{x} + \mathbf{n}_i$ at discrete time $t + 1$.

The *neighborhood* of a site at position $\mathbf{x}$ is the set of q sites at positions $\mathbf{x} + \mathbf{n}_i$ with $i = 0, 1, \ldots, q-1$. The neighborhood of the site may contain the site itself if one of the neighborhood vectors is equal to the zero vector (noted $\mathbf{0}$).

Lattices used in the context of LB methods are generally classified using the notation DdQq where d is the number of dimensions and q the number of velocity vectors. Note that for all lattices used in LB methods, for any velocity $\mathbf{v}_i$, $\mathbf{v}_j$ exists such as $\mathbf{v}_i + \mathbf{v}_j = \mathbf{0}$. Similarly, for any neighborhood vector $\mathbf{n}_i$, $\mathbf{n}_j$ exists such as $\mathbf{n}_i + \mathbf{n}_j = \mathbf{0}$.

Figure 1 shows a site of a D3Q19 lattice and its neighbors. On the figure, only 18 neighborhood vectors are represented. The last neighborhood vector being zero vector: the site is also a neighbor of itself.

LB methods are based on the following equation:

$$f_i(\mathbf{x} + \mathbf{n}_i, t+1) = f_i(\mathbf{x}, t) + \Omega_i \qquad i = 0, 1, \ldots, q-1 \tag{1}$$

where $\mathbf{x}$ is the position of a site, $\mathbf{n}_i$ a neighborhood vector and t discrete time. $f_i : \mathbb{Z}^3 \times \mathbb{Z} \to \mathbb{R}$ is called the *particle distribution function*.

$f_i(\mathbf{x}, t)$ is a real value from the interval [0..1], called *density*, that can be interpreted as the probability of having particles at position $\mathbf{x}$ moving along velocity vector $\mathbf{v}_i$ at time t. A velocity vector equal to zero vector therefore allows to represent particles at rest.

The term Ω_i is called *collision operator*. It describes the interaction of particles located at a point at a given time. A common method to compute Ω_i is the BGK model [9].

The collision operator is only applied on sites that are not obstacles. A widely used method called *bounce-back* [10] is applied otherwise. The main idea of the bounce-back is that a particle following a given direction and that collides with an obstacle "bounces" in the opposite direction. A more formal definition is given below.

The computation of the densities at time $t+1$ in function of densities at time t generally involves two main phases: *propagation* (also called streaming) and *collision*. Propagation is the operation of "moving" $f_i(\mathbf{x}, t)$ values to $\mathbf{x}$'s neighbors $\mathbf{x} + \mathbf{n}_i$ with $i = 0, 1, \ldots, q-1$. Collision is the computation of the new particle distribution functions values $f_i(\mathbf{x}, t+1)$ given the interaction of particles (and, therefore, based on propagated values). These two operations can be highlighted by decomposing Equation 1 into two equations, first representing propagation and second representing collision:

$$\hat{f}_i(\mathbf{x} + \mathbf{n}_i, t) = f_i(\mathbf{x}, t) \tag{2}$$

$$f_i(\mathbf{x}, t+1) = \hat{f}_i(\mathbf{x}, t) + \Omega_i \tag{3}$$

In case $s(\mathbf{x})$ is *true*, bounce-back is applied and following equation is used instead of Equation 3:

$$f_i(\mathbf{x}, t+1) = \hat{f}_j(\mathbf{x}, t) \tag{4}$$

where i and j are such as $\mathbf{n}_i = -\mathbf{n}_j$.

Note that most collision operators (BGK included) are local i.e. computed for each site in function of the densities associated to the site: $\Omega_i(\mathbf{x}, t) = g_i(\hat{\mathbf{f}}(\mathbf{x}, t))$ where $g_i : \mathbb{R}^q \to \mathbb{R}$ is a function that depends on the operator type.

A fluid simulated with numerical methods must be defined on a bounded spatial domain. Boundary conditions (periodic, pressure or velocity conditions) are therefore used to describe the fluid dynamics on its boundaries composed of the lattice borders.

A general definition for the *border* of a lattice is the set of sites that lack at least one neighbor: let $\mathbf{x}$ be the position of one of these sites, it exists a neighborhood vector $\mathbf{n}_i$ such as $\mathbf{x} + \mathbf{n}_i$ is not the position of a site of the lattice (the position is out of the lattice).

A 3D finite lattice (D3Qq family) is a cube or a cuboid. The components of position $\mathbf{x} = (x, y, z)$ have ranges defined in the following way: $0 \leq x < xSize$, $0 \leq y < ySize$ and $0 \leq z < zSize$ where $xSize$, $ySize$ and $zSize$ are the size of each dimension. The *size* of a 3D lattice is defined by the vector $(xSize, ySize, zSize)$ and the number of sites of the lattice is given by $xSize \times ySize \times zSize$.

A 4D array `f` of real values can therefore be used to represent the particle distribution functions of a lattice. In the same way, the solid function s can be represented by a 3D array of booleans.

These arrays can be declared as follows:

```
f :  array[0..xSize-1, 0..ySize-1, 0..zSize-1, 0..18] of real;
s :  array[0..xSize-1, 0..ySize-1, 0..zSize-1] of boolean;
```

where `xSize`, `ySize`, `zSize` are the components of the size of the lattice (in number of sites).

Below is a typical LB simulation code, `timeSteps` is the number of time steps of the simulation:

```
"Fill s";
"Initialize f";
t := 0;
do t < timeSteps →
   "Propagate values";
   "Apply boundary conditions";
   "Apply collision";
   t := t + 1
od
```

A naive implementation of "Propagate values", directly based on Equation 2, requires an additional array fHat declared in the same way as f and representing $\hat{f}$. "Apply collision" uses the values from fHat and stores them back to array f (Equation 4 if the site is an obstacle or Equation 3 if it is not).

However, technics exist [3, 4, 5, 7] that avoid the use of the fHat array and therefore almost divide memory usage by two. In particular, Murphy [7] suggests a method where densities are displaced in f in an order depending on the associated velocity vector and preventing the destruction of information. "Apply collision" can then directly modify the values of f.

For example, the in-place propagation of the densities associated to velocity vector $\mathbf{n}_8 = (-1, 1, 0)$ is described by following algorithm:

```
x, y, z := 1, ySize - 2, 0;
do x < xSize →
   f[x-1, y+1, z, 8] := f[x, y, z, 8];
   if z < zSize - 1 → z := z + 1
   □ z = zSize - 1 →
      z := 0;
      if y > 0 → y := y - 1
      □ y = 0 →
         y := ySize - 2; x := x + 1
      fi
   fi
od
```

where densities associated to velocity $\mathbf{n}_8$ are displaced using a scan that has a direction $(1, -1, 0)$ opposed to $\mathbf{n}_8$.

In general, the propagation of all densities associated to each velocity i is a translation following vector $\mathbf{n}_i$ and using a scan that has direction $-\mathbf{n}_i$. The propagation step for a D3Q19 lattice is therefore composed of 18 translations (translation following rest vector $\mathbf{n}_0$ has no effect).

3. Optimizing Propagation Step

Unlike other optimization technics [3, 4, 5], Murphy's method [7] does not merge collision and propagation step. It implies a propagation optimized data layout which is, in general, an approach that gives good results [6].

3.1. Data Locality and Representation of Multi-dimensional Arrays

Memory has a hierarchical organization [11]: data used as operands of an instruction are loaded into the registers from cache memory. If data are not available in cache memory, a *cache miss* occurs and the program's execution is interrupted until data are transferred from slow main memory to fast cache memory. When a cache miss occurs, a portion of main memory is copied into cache memory.

Typical code optimization includes the minimization of the number of cache misses by using an optimal data organisation in memory and an optimal data access "scheduling". The *data locality* principle (data brought into cache are used at least once before being flushed out of it) is then ensured.

LB simulations are memory intensive applications, ensuring the data locality principle is therefore essential and addressed by most optimized LB simulation algorithms [3, 4, 5].

In order to use the optimization proposed by Murphy [7], a multi-dimensional array needs to be represented by a 1D array (as done internally in languages supporting directly multi-dimensional arrays). Figure 3 shows the representation of array A from Figure 2 in a 1D array. Consecutive lines are then contiguous in memory which potentially ensures a better data locality when array is scanned line by line.

In general, for a dD array of size $(s_1, s_2, \ldots, s_d)$, a 1D array of size $\prod_{i=1}^{d} s_i$ is needed. An *index function* $\delta : \mathbb{N}^d \to \mathbb{N}$ that, from the position $(i_1, i_2, \ldots, i_d)$ (with $0 \le i_j < s_i$ for $i = 1, 2, \ldots, d$) of an element of the dD array, gives the position in the 1D array must also be defined. In the example given in Figure 3, $\delta(i_1, i_2) = i_1 \times 3 + i_2$ with $0 \le i_1, i_2 < 3$.

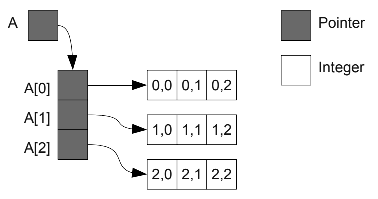

Figure 2: 2D array A of 3×3 integers represented using arrays of arrays.

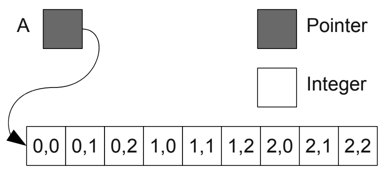

Figure 3: 2D array of 3×3 integers represented using a 1D array.

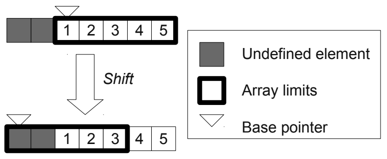

Figure 4: Array shift with base pointer move.

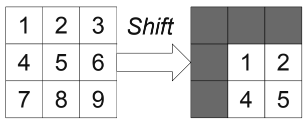

Figure 5: 2D array shift using offset vector (1, 1).

Let vf be the 1D representation of a D3Q19 lattice of size $(xSize, ySize, zSize)$. The size of vf (in number of floating point values) is given by $xSize \times ySize \times zSize \times 19$. To have the same data organization as used in simple implementation of Section 2, the following index function must be used:

$$\delta(x, y, z, q) = x \times (ySize \times zSize \times 19) + y \times (zSize \times 19) + z \times 19 + q$$

With this representation, in order to ensure data locality, lattice elements should be accessed in the following way:

```
x, y, z, q := 0, 0, 0, 0;
do x < xSize →
   "Access vf[delta(x, y, z, q)]";
   if q < 18 → q := q + 1
   □ q = 18 →
      q := 0;
      if z < zSize - 1 → z := z + 1
      □ z = zSize - 1 →
         z := 0;
         if y < ySize - 1 → y := y + 1
         □ y = ySize - 1 →
            y := 0; x := x + 1
         fi
      fi
   fi
od
```

Note that the implementation of in-place propagation described in Section 2 does not necessarily ensure data locality.

3.2. Array Shift

Murphy's method [7] is based on the constant time shift of elements of an array. The shift operation is implemented by moving the base pointer (pointer to the first element of the array) of the array to the "right" (base pointer is increased) or to the "left" (base pointer is decreased).

For example, Figure 4 illustrates the shift of an array containing five elements two positions to the right. The base pointer of the array is moved two positions to the left. As shown in the figure, an element at position i in the array before shift has position $i + 2$ after shift.

With a shift of n positions to the right, n elements are undefined at the beginning of the array. With a shift of n positions to the left, n elements are undefined at the end of the array.

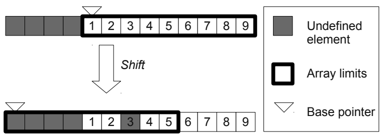

Figure 6: Shift of the vector representation of a (3 × 3) array.

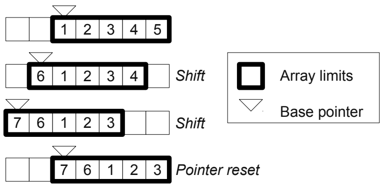

Figure 7: Two left shifts and a pointer and data reset.

The shift of the elements of an array is defined by an integer called the *offset*: with an offset equal to n, after the shift, the element at position i is at position $i + n$ if it is in the array boundaries. If n is positive, elements are shifted to the right. If n is negative, elements are shifted to the left.

The shift of the elements of a dD array is defined by a vector called *offset vector*: with an offset vector $\mathbf{v}$ an element at position $\mathbf{p}$ has position $\mathbf{p} + \mathbf{v}$ after the shift if it is in the array boundaries. Figure 5 illustrates the shift of the elements of a 2D array of size $(3, 3)$ with an offset vector $(1, 1)$.

If a dD array is represented using a 1D array, the shift of its elements using vector $\mathbf{v}$ is obtained by shifting the 1D array using offset $\delta(\mathbf{v})$. Figure 6 illustrates the shift of the elements of the 1D array backing the 2D array of Figure 5 with offset $\delta(1, 1) = 1 \times 3 + 1 = 4$. In the figure, the element with label “3” has been marked as undefined. Its value is, in fact, known but does not make sense in the context of the multi-dimensional array shift.

Note that in order to use the base array pointer move method to shift elements of an array without potentially overwriting required data in memory, enough memory needs to be reserved to the left and/or to the right of the array.

3.3. Propagation Based on Array Shifting

In-place propagation presented in Section 2 can be implemented using multi-dimensional array shifting if lattice values are reorganized: the values associated to each velocity should be grouped in separate 3D arrays, each represented by a 1D array. Instead of a 4D array, a D3Q19 lattice is then represented by 19 1D arrays. The index function $\delta(x, y, z)$ for these arrays is given by:

$$\delta(x, y, z) = x \times (ySize \times zSize) + y \times (zSize) + z$$

As stated previously, propagation is the application of a translation vector to all values associated to each velocity. Propagation can therefore be implemented by shifting each of the 19 arrays of new representation by a fixed offset.

The offset d_i to apply to the array associated to velocity i is given by $\delta(\mathbf{n}_i)$ where $\mathbf{n}_i$ is the neighborhood vector associated to velocity i.

Memory must be reserved around the array in order to move the array base pointer without overwriting other used data. In its implementation, Murphy allocates a supplementary space of $|d_i| \times k$ values. This way, k shifts can occur before no more space is available to continue to shift array base pointers. At this point, array elements are copied back to their initial positions and the base pointer is reset. k new shifts can then occur again. Figure 7 shows an array of five elements with 2 additional positions reserved at its left. Two right shifts with an offset of 1 can then be performed before pointer and data are reset.

3.4. Circular Array Shifting

To avoid the memory overhead and pointer and data reset operation of Murphy’s method, we use a *circular array*: the base pointer never moves but an offset pointing to the actual first position of the circular array can be moved (see Figure 8).

A circular array represented using a simple array can be declared as follows:

```
var
  v :  array[0..N-1] of "type";
  off :  integer
```

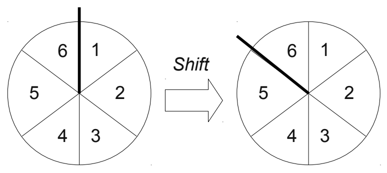

Figure 8: Circular array and one position right shift.

where v contains the data of the circular array, N is the size of the circular array and off the index of the first element of the array. The element at position i in the circular array is at position $(off + i)$ mod N in v. To apply an offset a to the elements of v, the following command is executed: `off := (off - a) mod N`.

In an implementation of propagation using circular array shift, a D3Q19 lattice can be represented as follows:

```
var
   f :  array[0..18, 0..M-1] of real;
   offs :  array[0..18] of integer
```

where f is a 2D array of real values: each line f[i,.] corresponds to the 1D representation of a 3D array containing the densities associated to velocity i with $0 \leq i < 19$; offs is an array of integers that contains the current offset associated to each line of f.

Propagation based on circular array shifts is more efficient than Murphy's method because there is no substantial memory overhead and no "pointer and data reset" operations are needed. However, the cost of accessing elements of the array is increased because of the additional modulus. This cost is discussed in next section.

4. Adapting Collision to New Data Organization

The new data organization dramatically improves the efficiency of propagation operation, however, due to data locality problems and a more complex element access, slows down collision step.

Let xyzSize be the size of each line f[i, .]. The 19 densities f[i, (k + offs[i]) mod xyzSize] with $0 \leq i < 19$ are associated to the same site. Accessing these 19 densities subsequently as it is currently done in collision step does probably not ensure data locality as these values are "far" regarding their memory address. The number of cache misses is therefore not minimized. However, accessing subsequently b densities f[i, (k + offs[i]) mod xyzSize] with i fixed and $k \in [a..a + b[$ ensures a better data locality as these densities are contiguous in memory.

To reduce the number of cache misses and complex element accesses, the values associated to a part of lattice sites can be copied into a small 2D array, called *block*, that fits entirely in cache memory. A block has B lines and 19 columns: B is the maximum number of sites the block can contain and each line contains the densities of a site. The operation of copying densities from lattice to a block, called *lattice-block copy*, must be written in a way that minimizes the number of cache misses. Collision can then be applied on block's sites and the lattice updated with block's data (*block-lattice copy*). As for lattice-block copy, block-lattice copy must be implemented in a way that minimizes the number of cache misses.

A block can be declared as follows:

```
type
   Block = record
      size :  integer;
      data :  array[B, 0..18] of real;
      xPos :  array[B] of integer;
      yPos :  array[B] of integer;
      zPos :  array[B] of integer
   end
```

where `B` is the maximum size of the block, `size` its actual size, `data` the array containing densities and `xPos`, `yPos` and `zPos` arrays containing respectively the x, y and z components of the extracted sites' positions in the lattice.

The collision step (`"Apply collision"`, see Section 2) can be rewritten in order to access lattice's sites block by block and therefore ensuring better data locality in the context of the lattice representation introduced in Section 3.4. Next algorithm describes the rewritten collision step using block access:

```
var
   block : Block;
   k : integer
begin
   k := 0;
   do k < xyzSize →
      latticeBlockCopy(k, block);
      "Apply collision on copied block";
      blockLatticeCopy(k, block);
      k := k + block.size
   od
end
```

`"Apply collision on copied block"` simply consists in applying collision operator or bounce-back to each site of the block.

As suggested above, `latticeBlockCopy` and `blockLatticeCopy` procedures copy densities from lattice to block and vice-versa by accessing subsequent densities velocity per velocity. In addition, the use of the modulus to compute the position of a density in each line `f[i,.]` array is minimized in order to reduce the access overhead caused by circular array representation. This is done by computing the position `from` of the first density to copy and the position `to` such as (`to` -1) is the position of the the last density to copy from a line `f[i,.]` into a block. As `f[i, .]` represents a circular array, `to` may be lower than `from`. In this case, the densities to copy are in two areas: `f[i, from..xyzSize-1]` and `f[i, 0..to-1]`. Otherwise, the densities to copy are in one area: `f[i, from..to-1]`.

The procedure `latticeBlockCopy` is described below. The parameter `start` is the position of the first density to copy from each circular array and `bl` is the destination block. A precondition on `start` is that $0 \leq$ `start` $<$ `xyzSize` i.e. `start` must point a value of a line of `f`.

```
procedure latticeBlockCopy(start : integer;
   var bl : Block);
begin
   bl.size := min(xyzSize - start, B);
   if bl.size = 0 → skip
   □ bl.size > 0 →
      q := 0;
      do q < 19 →
         from := (offs[q] + start) mod
         xyzSize;
         to := (offs[q] + start + bl.size)
            mod xyzSize;
         "Copy of densities from lattice
            to block";
         q := q + 1
      od;
      "Set positions in bl"
   fi
end
```

```
** Copy of densities from lattice to block **
if from ≤ to →
   pos, i := q, from;
   do i < to →
      bl.data[pos] := f[q, i];
      i, pos := i + 1, pos + 19
   od
□ from > to →
   pos, i := q, from;
   do i < xyzSize →
      bl.data[pos] := f[q, i];
      i, pos := i + 1, pos + 19
   od;
   i := 0;
   do i < to →
      bl.data[pos] := f[q, i];
      i, pos := i + 1, pos + 19
   od
fi
```

Finally, the positions of the sites in the block are computed (`"Set positions in bl"`). The use of the modulus is avoided as much as possible. The position of the first site of the block is computed in function of `start` and the positions of next sites are set accordingly. The algorithm below describes this operation:

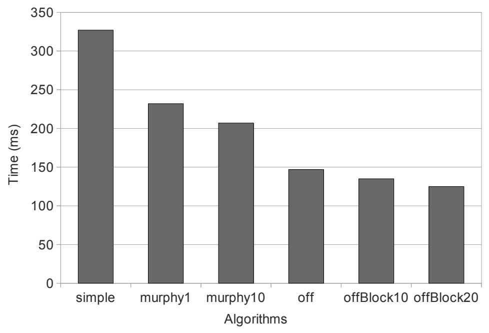

Figure 9: Comparison of execution time for a complete simulation when using different algorithms.

```
yzSize := ySize * zSize;
x := (start div yzSize);
y := (start mod yzSize) div zSize;
z := start mod zSize;
site := 0;
do site < bl.size →
   xPos[site], yPos[site], zPos[site] := x, y, z;
   site := site + 1;
   if z < zSize - 1 → z := z + 1
   □ z = zSize - 1 →
      z := 0;
      if y < ySize - 1 → y := y + 1
      □ y = ySize - 1 →
         y := 0; x := x + 1
      fi
   fi
od
```

The principle behind block-lattice copy is similar to lattice-block copy.

Note that "Apply boundary conditions" should also be adapted to circular array-based lattice representation by following above principles (maximizing data locality and avoiding as much as possible the use of the modulus to compute the position of a density).

5. Comparison of Simple and Optimized Algorithms

In order to evaluate our optimization method, we applied it to an existing implementation that is part of a tool that aims to be highly portable and is therefore written in Java. This explains the relatively poor performance of the implementation when compared to optimized native codes. However, we should be able to observe the execution time reduction implied by our optimization method.

Figure 9 shows the execution times for one time step of a complete simulation (propagation, boundary conditions and collision) on a $(64, 64, 64)$ D3Q19 lattice. These execution times have been measured on a computer equipped with an Intel® Core™ 2 2.13Ghz CPU with 2 MBytes of cache memory. The labels of the figure's legend have the following meaning:

- simple: Simple lattice representation described at the end of Section 2 with in-place propagation;
- murphyα: Murphy's optimization with data and pointer reset operations required every α simulation iterations;
- off: Circular array representation, shift based propagation, no block access;

- offBlockβ: Circular array representation, shift based propagation, block access (maximum block size $B = \beta$);

The use of circular arrays in order to avoid data and pointer reset operations increases simulation code efficiency when compared to Murphy's method. Also, "murphy10" implementation is 1.6 times faster than "simple" implementation which is what Murphy observed with its Fortran implementation [7]. Finally, block access further improves efficiency and "offBlock20" implementation is more than 2.5 times faster than "simple" implementation.

Furthermore, we observe that the Java implementation used to produce above results is able to handle more than 2 million sites per second ("offBlock20" implementation). This is "only" 2 times slower than native optimized implementations on almost similar equipments (AMD® Opteron™ 2Ghz and Intel® Xeon™ 3.2Ghz) [12]. We therefore expect to reach at least similar results if applying our optimization technic to a native implementation.

6. Conclusion

LB methods require large amounts of memory and computational power. In order to minimize these requirements, an optimized implementation is required. LB methods are memory intensive applications; a proper representation of these arrays and ensuring data locality are therefore essential.

Murphy [7] provides an interesting data layout and an implementation (see Section 3) that minimizes propagation step execution time. In particular, his method is based on constant time array shifting that can be used to implement the propagation step. This constant time array shifting method implies the displacement of the array base pointer and therefore requires additional memory reserved before or after the array. In addition, data need to be regularly copied back to their original position (pointer and data reset operation).

In Section 3.4, we propose to use a constant time circular array shifting. With this method, there is no more need to reserve memory before or after the array and pointer and data reset operations can be avoided. However, our method implies a more complex access to the elements of the array and, in the same way Murphy's method does, does not ensure data locality for collision step.

To handle this issue, we introduced block access in Section 4: blocks containing the state of several sites are copied from lattice in a way that minimizes cache misses, collision is then applied on block's sites and block's data are copied back into the lattice, again by ensuring best data locality. This process is repeated until all sites have been collided.

Finally, we observed in Section 5 that our enhanced optimization applied to an existing Java implementation of LB simulations is more than 2.5 faster than the simple implementation with a $(64, 64, 64)$ D3Q19 lattice on a computer with an Intel® Core™ 2 2.13Ghz CPU with 2 MBytes of cache memory allowing to handle more than 2 million sites per second.

In future work, we plan to use our method to optimize a native implementation of LB simulation code and compare it to other "state of the art" optimization technics.

7. References

[1] S. Chen, G. D. Doolen, Lattice Boltzmann method for fluid flows, Annual Review of Fluid Mechanics 30 (1998) 329–364.

[2] S. Succi, The Lattice Boltzmann Equation for Fluid Dynamics and Beyond, Oxford University Press, 2001.

[3] J. Latt, How to implement your DdQq dynamics with only q variables per node (instead of 2q), Tech. rep., Tufts University (June 2007).

[4] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, J. Westerholm, An efficient swap algorithm for the lattice Boltzmann method, Computer Physics Communications 176 (3) (2007) 200–210.

[5] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, U. Rüde, Optimization and profiling of the cache performance of parallel lattice Boltzmann codes, Parallel Processing Letter 13 (4) (2003) 549–560.

[6] G. Wellein, T. Zeiser, G. Hager, S. Donath, On the single processor performance of simple lattice boltzmann kernels, Computers & Fluids 35 (8-9) (2006) 910–919.

[7] S. Murphy, Performance of Lattice Boltzmann kernels, Master's thesis, University of Edinburgh, this document is available at `http://www2.epcc.ed.ac.uk/msc/dissertations/dissertations-0405/0762240-9j-dissertation1.1.pdf` (2005).

[8] E. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Communications of the ACM 18 (1975) 453–457.

[9] P. L. Bhatnagar, E. P. Gross, M. Krook, A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems, Physical Review 94 (1954) 511–525.

[10] S. Wolfram, Cellular automaton fluids 1: Basic theory, Journal of Statistical Physics 45 (3-4) (1986) 471–526.

[11] S. Goedecker, A. Hoisie, Performance Optimization of Numerically Intensive Codes, SIAM, 2001.

[12] K. Mattila, J. Hyväluoma, J. Timonen, T. Rossi, Comparison of implementations of the lattice-Boltzmann method, Computers and Mathematics with Applications 55 (2008) 1514–1524.