FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Implementation and Optimization of a Cache Oblivious Lattice Boltzmann Algorithm

Aditya Nitsure

Master Thesis

Implementation and Optimization of a Cache Oblivious Lattice Boltzmann Algorithm

Aditya Nitsure

Master Thesis

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Klaus Iglberger (M. Sc.),
	Dr. Gerhard Wellein,
	Dr. Georg Hager
Bearbeitungszeitraum:	1.2.2006 – 31.7.2006

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 31. July 2006 ....

Acknowledgments

At the completion of my thesis, I am filled with satisfaction and the feeling of achievement. Most of the work presented in this thesis has been a team effort. I would like to thank those with whom I collaborated throughout my thesis work.

I am extremely grateful to Prof. Dr. Ulrich Rüde for considering me capable enough, offering me the thesis at his group and helping me both in technical and nontechnical matters throughout the thesis period. I am thankful to him for giving me the freedom of working time.

I would like to convey my special thank to Klaus Iglberger (M. Sc.), for being a guide. I admire his promptness and style of working.

I am highly thankful to Dr. Gerhard Wellein and Dr. Georg Hager for helping me out in complex things related to optimization and giving me privilege to work on latest systems at RRZE .

I would like to thank Dipl.-Inf. Christian Feichtinger for helping me through the final corrections of the thesis.

I would like to express my deep appreciation to Dr. Reinhard Hübner for offering me a lot of flexibility while working with him during my study period.

Thanks to all friends from Erlangen especially Gagan, Kunal and Shivendra with whom I have enjoyed numerous scientific and non-scientific discussions throughout the years.

Last but not least, I would like to express my deep gratitude to my dear parents and all my friends from India. Thanks for your invaluable support!

Contents

Zusammenfassung

Die Lattice Boltzmann Methode (LBM) ist eines der neueren Verfahren um Flüssigkeiten zu simulieren. Eines der momentanen Forschungsgebiete im Bereich der Lattice-Boltzmann Methode ist die effiziente Implementierung des LBM auf Cache basierten Microprozessoren. Dabei ist die resultierende Performance nicht nur von Prozessortakt abhängig, sondern ebenfalls von der Ausnutzung der Cachehirarchie. Eine Möglichheit diese zu optimieren ist Blocking , welches speziell auf eine Cachearchitektur angepasst werden muss.

In dieser Arbeit wird ein Verfahren zur effizienten Nutzung des Cache vorgestellt, dass unabhängig von der jeweilige Cachearchitektur ist. Dabei lag das Ziel dieser Arbeit in der Implementierung und Optimierung des *Cache Oblivious Lattice Boltzmann Algorithm(COLBA)*, welcher auf einem Cache unabhängigen stern basierenten Verfahren von Frigo aufbaut. Ausserdem wurde die Performance des COLBAs mit der Performance von bereits existierenden LBM Implementierangen verglichen. Hierbei wurde gezeigt, da die Performance des COLBAs mit dem der existierenden, auf Blocking basierenden Implementierungen zu vergeichen ist. Des weiteren wurde das parallele Verhalten der Methode untersucht.

Abstract

The Lattice Boltzmann Method (LBM) is one of the modern techniques to simulate fluids. A lot of work has already been done to speed up the LBM on cache-based microprocessor architecture. In order to achieve high performance on these architectures, not only the clock rate of the processor plays an important role, but also the significant influence of the caches have to be taken into account. A common method to exploit spatial and temporal data locality by reusing the cache is blocking (a cache aware algorithm). In this thesis, a *Cache Oblivious Lattice Boltzmann Algorithm (COLBA)* and its optimization techniques are proposed.

The main aim of thesis has been to test the practicality of Frigo's stencil based cache oblivious algorithm for the LBM and to compare the performance of the cache oblivious algorithm and cache aware / blocked algorithms on different architectures. Additionally, the feasibility of COLBA in parallel environment and its scalability were tested.

In this thesis, results of the COLBA for a single processor and multi-processor parallel version are presented. COLBA has shown equivalent performance to the efficient LBM kernel at RRZE. Additionally the performance of the COLBA is compared for different architectures.

Part I

Introduction

Chapter 1

Introduction

Since the past decade, cache-based microprocessors and hierarchy based memory structures are very common in the area of High Performance Computing. Despite improvements in technology, microprocessors are still much faster than the main memory. Therefore memory access time is increasingly the bottleneck in overall application performance. To close the gap between processor speed and memory bandwidth, modern architectures use hierarchies of small but fast caches. Blocking is one of the cache based approaches of performance optimization which reduces the data transfer from/to main memory by increasing spatial and temporal locality i.e. reusing the cached data again and again.

In the recent computational development of the Lattice Boltzmann Method, blocking has been used to improve the computational performance [4]. Normally, for effective blocking, the cache parameters like cache size, cache length etc. are considered. In this thesis work, a *Cache Oblivious Lattice Boltzmann Algorithm* for cache based architectures is introduced, which is independent of the cache parameters.

The main idea of the cache oblivious algorithm has been introduced by Matteo Frigo [3]. Previously, the cache oblivious algorithm has been found to be effective for problems like matrix multiplication, FFT, sorting (merge sort) and matrix transpose. It was also observed by Prokop [7] for the mentioned problems that the cache oblivious algorithm executes in less than 70% of the time of the iterative algorithm for problem sizes that do not fit in L2-cache. It has also been shown that the comparison between blocked and cache oblivious algorithms shows equivalent performance.

1.1 Introduction to the Lattice Boltzmann Method

The lattice Boltzmann Method (LBM) is a discrete computational method based on the Boltzmann equation and an alternative fluid simulation approach to the classical Navier-Stockes equation. The method simulates fluid by an estimation of the fluid particle distribution in each cell of the discretized simulation domain. The fluid particles move according to the particle velocity distribution functions into the neighboring cells and collide with all other particles, that stream into this cell from different direction. The Boltzmann equation is a statistical equation based on gas dynamics, which describes the probability that fluid particles with velocity ξ can be monitored at position x at time t. This is represented by the single particle distribution function $f\left(x, \vec{\xi}, t\right)$, where $\vec{\xi}$ is the particle velocity, in phase space $\left(x, \vec{\xi}\right)$ and time t. The kinetic model using the Boltzmann equation with the single relaxation time approximation (BGK model) is [6]:

$$\frac{\partial f}{\partial t} + \vec{\xi}\frac{\partial f}{\partial x} = -\frac{1}{\tau}\left[f - f^{(0)}\right] \tag{1.1}$$

where $f^{(0)}$ is the equilibrium distribution function (the Maxwell-Boltzmann distribution function) used to calculate the new values for the particle distribution function. To solve for f numerically, Eq. 1.1 is first discretized in the velocity space $\vec{\xi}$ using a finite set of velocities $\{\vec{\xi_\alpha}\}$:

$$\frac{\partial f_\alpha}{\partial t} + \vec{e_\alpha}\frac{\partial f_\alpha}{\partial x} = -\frac{1}{\tau}\left[f_\alpha - f_\alpha^{(0)}\right] \tag{1.2}$$

where $f_\alpha\left(x,t\right) = f\left(x, \vec{\xi_\alpha}, t\right)$ and $f_\alpha^{(0)}\left(x,t\right) = f^{(0)}\left(x, \vec{\xi_\alpha}, t\right)$ are the distribution function and the equilibrium distribution function of the α-th discrete velocity $\vec{\xi_\alpha}$, respectively.

For 2D flows, a 2-D square lattice model, denoted as D2Q9 model is widely used while for 3D flows, 3-D cubic lattice models, denoted as D3Q15, D3Q19, D3Q27 are used. These models are briefly explained in chapter 2. For athermal fluids, the equilibrium distributions for D2Q9, D3Q15, D3Q19, and D3Q27 models are all of the form :

$$f_\alpha^{(eq)} = \omega_\alpha \rho\left[1 + \frac{3}{c^2}\left(\vec{e_\alpha}\cdot\vec{u}\right) + \frac{9}{2c^4}(\vec{e_\alpha}\cdot\vec{u})^2 + \frac{3}{2c^2}\left(\vec{u}\cdot\vec{u}\right)\right] \tag{1.3}$$

where ω_α is a weighting factor and $\vec{e_\alpha}$ is a discrete velocity. The mass density $\rho\left(t,x\right)$ and the momentum density $\rho\left(t,x\right)u\left(t,x\right)$ are evaluated by the following formulae Eq. 1.4 and Eq. 1.5. [9].

$$\rho = \sum_\alpha f_\alpha \tag{1.4}$$

$$\rho\vec{u} = \sum_\alpha \vec{e_\alpha} f_\alpha \tag{1.5}$$

The weighting factors ω_α for D2Q9 model and D3Q19 are given below respectively :

$$\omega_\alpha = \begin{cases} \frac{4}{9} & \alpha = \text{C} \\ \frac{1}{9} & \alpha = \text{N,E,S,W} \\ \frac{1}{36} & \alpha = \text{NE,SE,SW,NW} \end{cases} \qquad \omega_\alpha = \begin{cases} \frac{1}{3} & \alpha = \text{C} \\ \frac{1}{18} & \alpha = \text{N,E,S,W,T,B} \\ \frac{1}{36} & \alpha = \text{NE,SE, SW, NW,BE,BW,BN,BS,TE,TW,TN,TS} \end{cases}$$

where the capital letters indicate a discrete velocity : C(center), N(north), S(south), W(west), E(east), T(top), B(bottom), NW(North-West), NE(North-East), SW(South-West), SE(South-East), TN(Top-North), TW(Top-West), TS(Top-South), TE(Top-East), BE(Bottom-East), BW(Bottom-West), BN(Bottom-North), BS(Bottom-South).

Eq. 1.2 is often discretized in space x and time t into

$$f_\alpha\left(\vec{x}+\vec{e_\alpha}\Delta t, t+\Delta t\right) = f_\alpha\left(\vec{x}, t\right) - \frac{\Delta t}{\tau}\Big[f_\alpha\left(\vec{x}, t\right) - f_\alpha^{(0)}\left(\vec{x}, t\right)\Big], \tag{1.6}$$

where $\vec{x}$ is now a cell in the discretized simulation domain (square cells in 2D and cubes in 3D), t is the current time step and $t+\Delta t$ the next time step. Eq. 1.6 shows the simplified LBE. The difference between the old value and the new, already streamed value is the weighted summation of the equilibrium distribution $f_\alpha^{(0)}\left(\vec{x}, t\right)$ and the non-equilibrium distribution $f_\alpha\left(\vec{x}, t\right)$. To facilitate the implementation of the LBM, Eq. 1.6 can be separated into two steps, known as the collide step and the stream step, shown in Eq. 1.7 and 1.8 :

$$\tilde{f_\alpha}\left(\vec{x}, t+\Delta t\right) = f_\alpha\left(\vec{x}, t\right) - \frac{\Delta t}{\tau}\Big[f_\alpha\left(\vec{x}, t\right) - f_\alpha^{(0)}\left(\vec{x}, t\right)\Big] \tag{1.7}$$

$$f_\alpha\left(\vec{x}+\vec{e_\alpha}\Delta t, t+\Delta t\right) = \tilde{f_\alpha}\left(\vec{x}, t+\Delta t\right) \tag{1.8}$$

where $\tilde{f}\left(t, \vec{x}\right)$ denotes the post-collision state of the distribution function. Eq. 1.7 can be interpreted as the collide step : the particles that are currently at lattice site $\vec{x}$ at time step t interact with one another, affecting the particle distribution in the next time step $t+\Delta t$. This interaction is modeled by the right hand side of the equation. Eq. 1.8 can be interpreted as the stream step : the flow of the particles from one lattice site to the next. The particles described by the value of the distribution function $\tilde{f_N}\left(\vec{x}, t+\Delta t\right)$ will flow to the site in the north of the current site, described by $f_N\left(\vec{x}+\vec{e_N}\Delta t, t+\Delta t\right)$, the value of the distribution function at E will flow to the site in the east, and so on for each direction except C, which remain at the current lattice site.

1.2 Cache Based Memory Hierarchy

A cache is a small high-speed buffer memory between the processor and the main memory. Generally, the cache memory is organized in multiple levels. Some of these levels may be a part of the microprocessor (L1 on-chip cache), whereas other levels are external to the chip. In general, the higher the level the farther away the cache is from the CPU, and the higher in capacity the slower it is to operate. The system first copies the data needed by the CPU from the main memory into the cache, and then from the cache into a register in the CPU. Storage of results is in the opposite direction. First the system copies the data into the cache. Depending on the cache architecture details, the data is then immediately copied back to memory (write-through), or deferred (write-back). If an application needs the same data again, data access time is reduced significantly if the data is still in the cache. Fig. 1.1 shows a cache based memory hierarchy in a contemporary computer architecture.

The operating cost of cache memories is defined in terms of Latency and Bandwidth. **Latency** is defined as the time it takes to fetch one unit of transfer (typically a cache line). **Bandwidth** is defined as the amount of data transfered between two cache units or cache

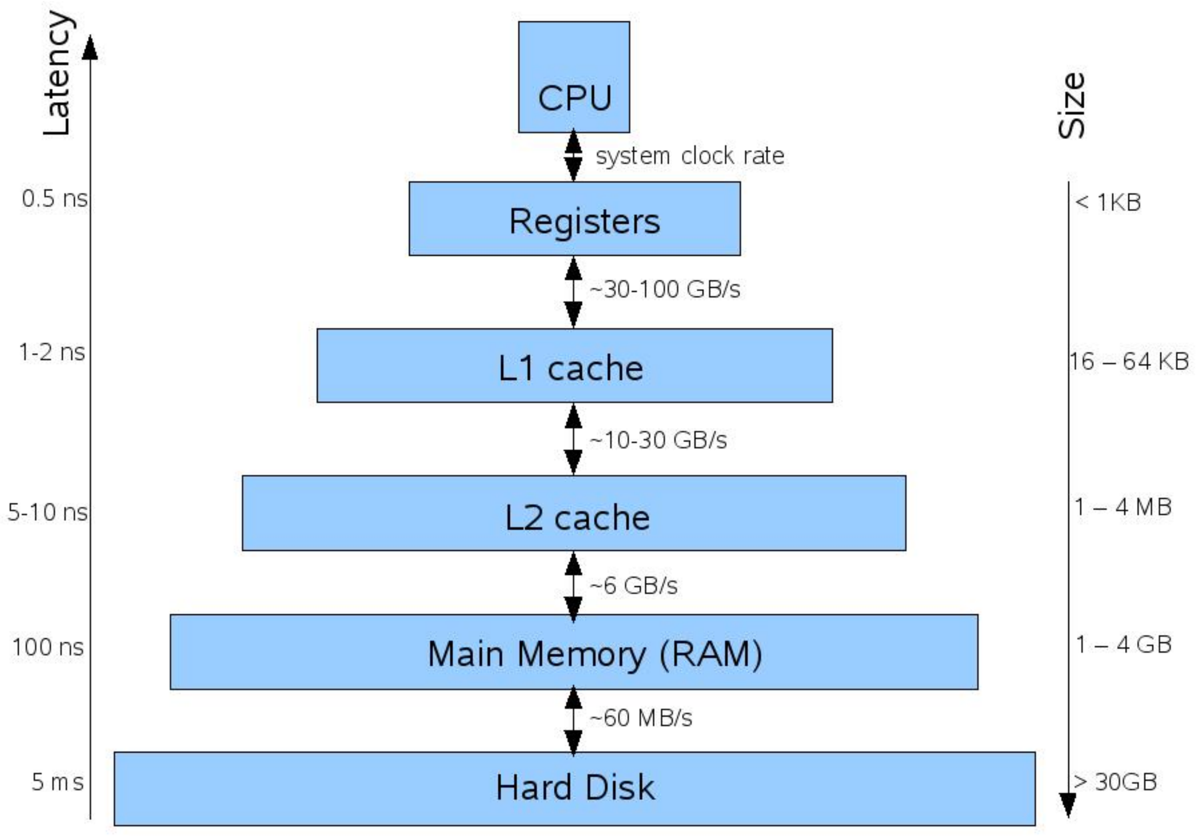


Figure 1.1: Typical memory hierarchy levels. Some modern computers have L3 level of cache between L2 cache and main memory.

and registers per unit time. As the speed of a component depends on its relative location in the hierarchy, the latency and bandwidth are not uniform.

Further, the on-chip caches are mainly classified as follows. They are not only varying in size and functionality, but also their internal organization is typically different.

Instruction Cache: The instruction cache is used to store instructions. This helps to reduce the cost of going to memory to fetch instructions. The instruction cache regularly holds several other things, like branch prediction information. In certain cases, this cache can even perform some limited operation(s).

Data Cache: A data cache is a fast buffer that contains the application data. Before the processor can operate on the data, it must be loaded from the main memory into the data cache in form of cache line. The element needed is then loaded from the data cache into a register and the instruction using this value can operate on it. The resulting value of the instruction is also stored in a register. The register contents are then stored back into the data cache. Eventually the cache line that this element is part of is copied back into the main memory.

TLB Cache: Translating a virtual page address to a valid physical address is rather costly. The TLB is a cache to store these translated addresses. Each entry in the TLB maps to an entire virtual memory page. The CPU can only operate on data and instructions that are mapped into the TLB. If this mapping is not present, the system has to re-create it, which is a relatively costly operation. The larger a page, the more effective capacity the TLB has. If an application does not make good use of the TLB (for example, random memory access) increasing the size of the page can be beneficial for performance, allowing for a bigger part of the address space to be mapped into the TLB.

Chapter 2

Cache Oblivious Algorithm

Cache Oblivious Vs Cache Aware Algorithms: A cache oblivious algorithm(COA) is defined as an algorithm independent of cache parameters e.g. cache size, cache length etc. In contrast to this, cache aware algorithms need cache parameters at run-time or compile-time for an optimal performance execution on a particular architecture. Blocking, for example, is one of the possible cache aware optimization technique. In this case, the block sizes depend on the actual cache size and/or cache line length. The common idea behind these algorithms is that both try to exploit *temporal* and *spatial* locality.

Temporal locality defines reusage of recently accessed data locations. It assumes that when one byte is accessed, it will be accessed again in the near future. **Spacial locality** defines the usage of data that is close to the recently accessed data location. It assumes that when a byte of data is requested by the CPU, subsequent requests will access neighboring bytes as well. Using spacial and temporal blocking most of the data needed by a process can be found in the cache so it increases the *cache hits* and reduces the *cache misses*. Indirectly this reduces the memory traffic between different levels of memory.

2.1 Stencil Based Cache Oblivious Algorithm

The stencil based COA has been introduced by Frigo [3]. Further the algorithm does work for from 1-dimensional to N-dimensional grids. For simplicity, COA is explained with the help of 1D space, a 1D time domain (2D space-time domain) and with a 3 point stencil calculation. Thus in the following example, a 2D discretized space-time rectangular domain has been used. The value at each point x_i at time t depends on x_{i-1}, x_i, x_{i+1} at time $t-1$.

The algorithm is based on a divide and conquer strategy. It recursively divides the space-time domain into smaller domains until it fits inside the cache and then solves the problem for the domain inside the cache. The complete rectangular space-time domain can be divided into triangles, parallelograms and trapezoids. Although we are interested in traversing a rectangular space-time domain, the algorithm can be explained with the help of trapezoidal region as shown in Fig. 2.1. The x and y axes represent space and time respectively. The complete trapezoid can be defined with parameters $t_0, t_1, x_0, x_1, \dot{x_0}, \dot{x_1}$. In physical terms t_0 and t_1 represent the start and end time step, x_0 and x_1 represent the start and end point of spatial domain and $\dot{x_0}$ and $\dot{x_1}$ represent the slop of the left and right edges of the trapezoid respectively. Thus $\mathcal{T}\left(t_0, t_1, x_0, \dot{x_0}, x_1, \dot{x_1}\right)$ represents a trapezoid. Any point inside this trapezoid can be defined as (t, x) such that $t_0 \leq t < t_1$ and $x_0 + \dot{x_0}\left(t - t_0\right) \leq x < x_1 + \dot{x_1}\left(t - t_0\right)$.

$\Delta t = \left(t_1 - t_0\right)$, represents the height of the trapezoid and $w = \left(x_1 - x_0\right) + \left(\dot{x_1} - \dot{x_0}\right)\frac{\Delta t}{2}$ represents the width of the trapezoid. We only consider a well defined trapezoid. i.e. the condition $t_1 \geq t_0$, $x_1 \geq x_0$, and $x_1 + \Delta t \cdot \dot{x_1} \geq x_0 + \Delta t \cdot \dot{x_0}$ holds.

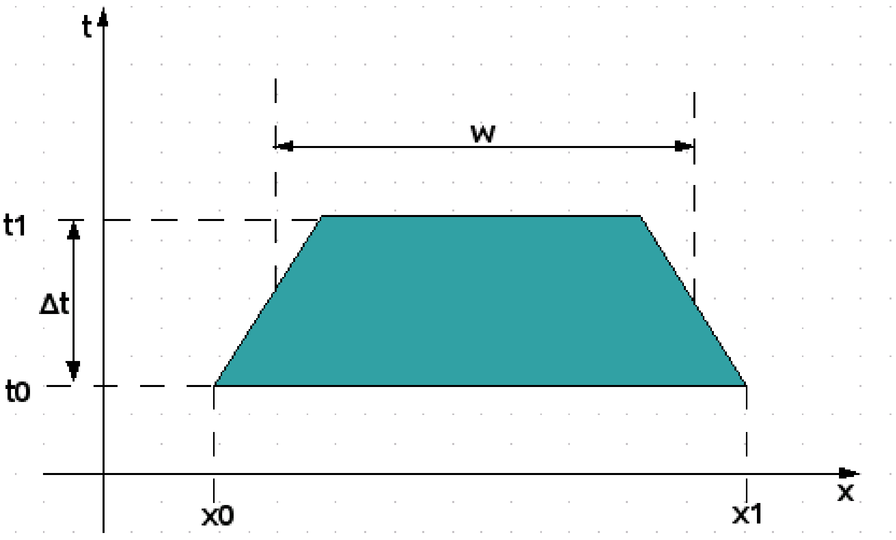


Figure 2.1: A trapezoidal region in space time domain.

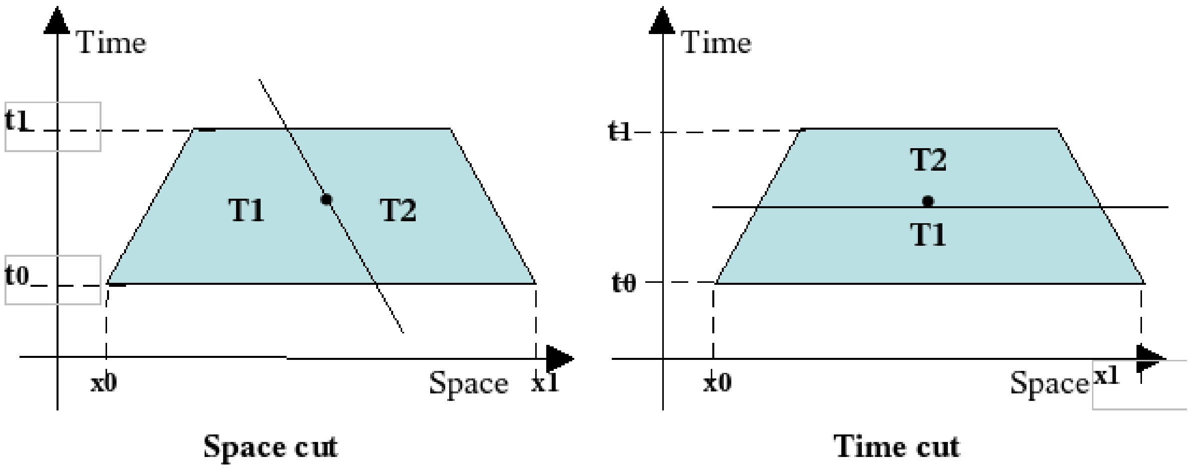


Figure 2.2: Illustration of space and time cuts, in a two dimensional trapezoid, consisting of a one dimensional space region and a one dimensional time region.

The algorithm visits all points of the trapezoid in an order that respects the stencil dependencies. It decomposes the trapezoid recursively into smaller trapezoids either by a space or time cut. The space cut is done, if the width is at least twice the height of the trapezoid. Then the trapezoid is cut along the line with slop -1 through the center of the trapezoid. If a space cut is not possible, a time cut is done along the horizontal line through center of trapezoid. Thus in both cases, two smaller trapezoids $\mathcal{T}_1$ and $\mathcal{T}_2$ are created as shown in Fig 2.2. The same procedure is applied on these newly created trapezoids until the problem is solvable inside the cache. The algorithm always first solves $\mathcal{T}_1$ and then $\mathcal{T}_2$. This traversal order is valid because no point in $\mathcal{T}_1$ depends on any point in $\mathcal{T}_2$.

Thus by changing the slope parameter, the same algorithm can be applied to domains of different shapes.

2.2 Ideal Cache Model

At this point, the Ideal Cache Model is introduced, which is used in [3] and [7] for the analysis of cache misses. The ideal cache model consists of the following assumptions:

1. **Optimal replacement** The replacement policy refers to the policy chosen to replace a block when a cache miss occurs and the cache is full. In most hardware, this is implemented as FIFO, LRU or Random. The model assumes that the cache line chosen for replacement is the one that is accessed farthest in the future. The strategy is called optimal *off-line* replacement strategy.

2. **2 levels of memory** There are certain assumptions in the model regarding the two levels of memory chosen. They should follow the inclusion property which says that data cannot be present at level i unless it is present at level i +1. In most systems, the inclusion property holds. Another assumption is that the size of level i of the memory hierarchy is strictly smaller than level i+1.

3. **Full associativity** When a block of data is fetched from the slower level of the memory, it can reside in any part of the faster level.

4. **Automatic replacement** When a block is to be brought in the faster level of the memory, it is automatically done by the OS/hardware and the algorithm designer does not have to care about it while designing the algorithm. Note that we could access single blocks for reading and writing in the external memory model, which is not allowed in the cache oblivious model.

In practice, it may not be possible to have an ideal cache as mentioned above. Typically, cache associativity is 2-way or 4-way for first level of cache and 8-way or 16-way for the next levels. As stated earlier, the replacement policy is operating system dependent, so the programmer has very limited control over these parameters. So it may decrease the performance of COA as compared to theoretical analysis. Furthermore, in the research work done by Frigo and Prakop, the same algorithm is justified for practical memory hierarchy systems.

Part II

Single Processor Implementation

Chapter 3

Cache Oblivious LBM Kernel Implementation

As the LBM is a major research area in the Department of System Simulation, sufficient amount of research has already been done on the efficient implementation of the LBM. The previous research from Jens Wilke [8], Klaus Iglberger [4] and Stefan Donath [2] was primarily focused on the choice of the underlying data structure and cache aware programming techniques.

For making the implementation more flexible, it was divided into two parts: the LBM implementation which is termed as LBM kernel and the implementation of the (COA). This offers the possibility to integrate any optimized new LBM kernel or completely different kernel without effort into the COA. In the following section, the implementation of the LBM kernel is discussed. After this, the implementation of the stencil based COA is presented and the integration of the LBM kernel is performed.

3.1 LBM Kernel in 2D

For the study of 2D COA, a 2D LBM was implemented using the popular D2Q9 model. This model consists of 9 velocities as follows: one at the center of the cell, where a velocity of zero can be assumed, the four velocities along the x and y axes in both directions respectively and four velocities along the diagonals of a square, in both directions. Fig. 3.1 shows the 2D lattice site of the D2Q9 model. The implementation of 2D LBM kernel is listed in App. B. The same 2D model was extended to 3D model. As the main focus was on the 3D implementation of the cache oblivious LBM, the discussion about 3D model will automatically cover all aspects of the 2D implementation and the optimization strategies.

3.2 LBM Kernel in 3D

A 3D lattice has to be chosen, consisting of 3D lattice sites, called cells, which contain the distribution functions of the particles. These distribution functions describe the distribution or velocity of the particles in each cell and therefore represents the flowing direction of the

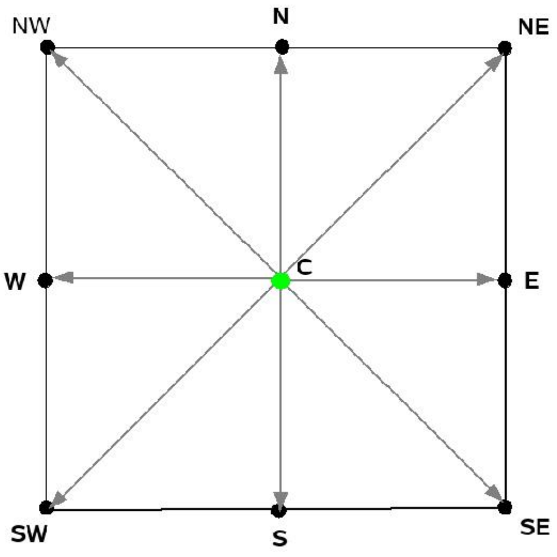


Figure 3.1: D2Q9 Model

discretized fluid. For the 3D case, three different models (D3Q27, D3Q19, D3Q15) are quite common.

In the D3Q27 model, every possible direction is used: one center, six for the axes in both directions respectively, twelve for all combinations of two axes and eight for all combinations of 3 axes. Though this model promises the best accuracy in terms of approximation and numerical stability, it is more computationally intensive. In contrast, the D3Q15 model consist only of 15 velocities(1 center, 6 for the axes and 8 for all combinations of 3 axes). Due to the small number of distribution functions, the D3Q15 model promises a very fast LBM, but this model is more prone to numerical instability and tends to anisotropic effects [6]. Fig. 3.3 shows D3Q15 and D3Q27 models.

The model used in this thesis is the D3Q19 model: it consists of 19 velocities and combines fast calculation and good approximation. It provides a balance between computational reliability and efficiency. The 19 velocities consist of velocity at the center of the cell, where a velocity of zero can be assumed, the six velocities along the x, y and z axes in both directions respectively and all combinations of two axes. Fig. 3.2 show D3Q19 model. For the LBM implementation the collide step equation Eq. 1.7 and the streaming step equation Eq. 1.8 are used. For each cell update, these two equations have to be calculated. In the streaming step, the velocities from the 18 surrounding cells are gathered and written to the cell being updated. The center does not have to be moved and represents the 19th velocity value. The collide step equation is executed afterwards and relaxes the distribution functions towards the local equilibrium f^{eq}. Fig. 3.4 illustrates the two steps of a cell update in a clearer 2D version: The first figure shows the streaming step, which results in an unbalanced particle distribution (second figure). The collide step takes the collision of the particle velocities into account and therefore relaxes the distribution towards the local equilibrium, which is represented by the circle.

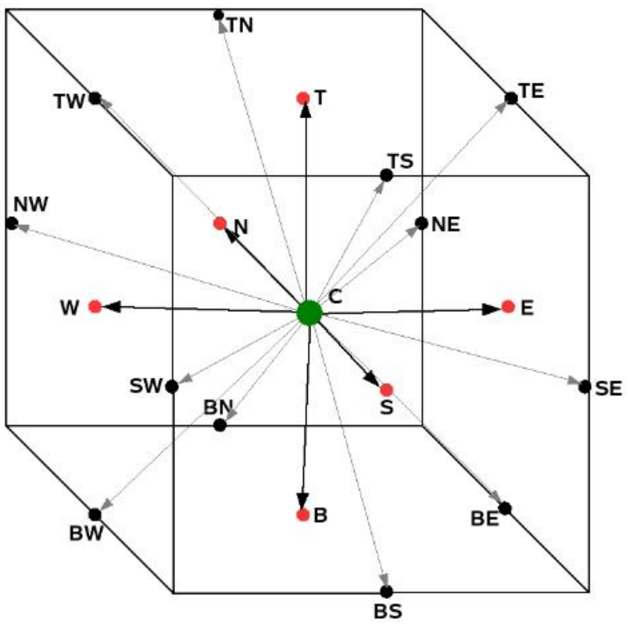


Figure 3.2: D3Q19 Model

Obviously, the order of the streaming and collide step can be reversed: in the order collide stream, the update of a cell would start by calculating the new distribution functions and end with spreading the new values to the surrounding neighbors. In contrast to the stream-collide order, where the old distribution functions are "pulled" into the updating cell from the surrounding neighbors, the newly calculated distribution functions are "pushed" to the neighboring cells.

3.3 Programming Frame Work

3.3.1 Review of Previous Work

As ample amount of research has been done previously, some of the conclusion are considered for this implementation.

1. A one dimensional array data layout is better, than a 3 dimensional array.
2. The stream-collide approach gives better performance for a non-blocked implementation than collide-stream approach.
3. The compressed grid certainly reduces the memory requirement compared to maintaining two separate arrays for the two grids.

3.3.2 Implementation Frame Work

Based on the previous results, the following implementation aspects were used:

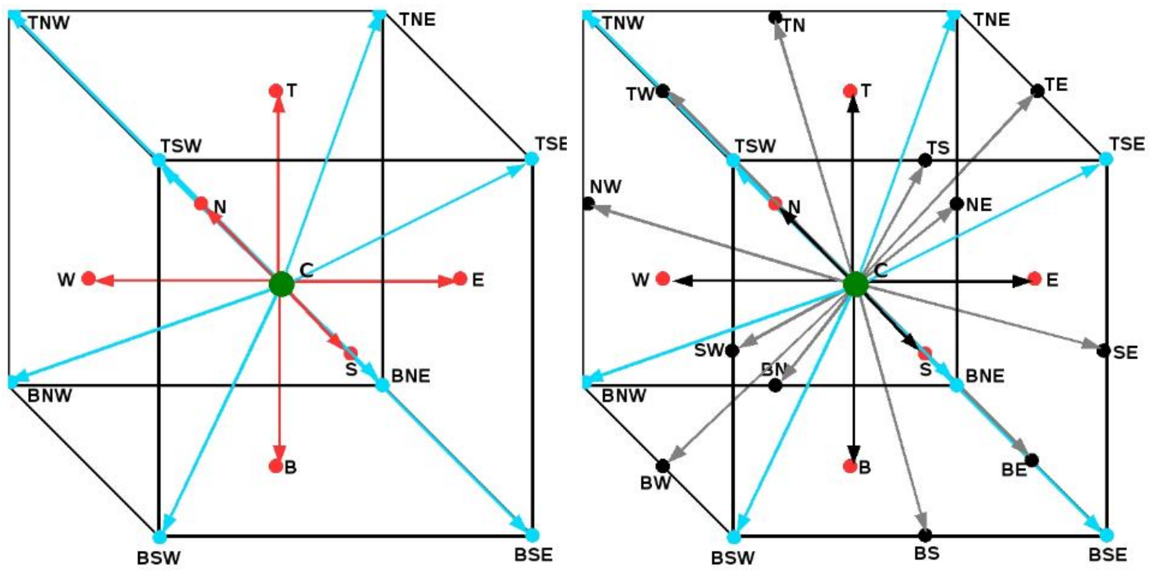


Figure 3.3: D3Q15 and D3Q27 Models

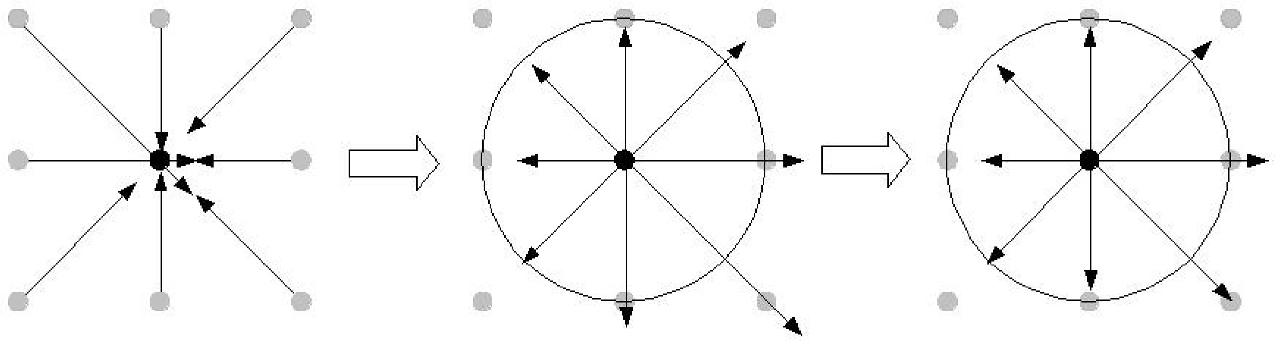


Figure 3.4: The stream and collide steps performed at one lattice site.

1. The stream-collide update order was chosen, because the focus was not on the traditional blocking implementation, so that the scheme which performed well for a non-blocked version was selected.

2. Two grids are used as the memory requirement is not an issue and no main focus of this thesis.

3. A 'Cell' structure is implemented to represent each lattice site, which holds 9 discrete velocities in 2D and 19 discrete velocities in 3D case. All the velocities are double precision values so the size of 'Cell' is 72 bytes (9 * 8 bytes) in case of 2D and 152 bytes (19 * 8 bytes) in case of 3D.

4. A one dimensional array of 'Cell' is used for the grid which have been mapped to three dimensional array using access macros.

All the implementation is done in the 'C' programming language. So having a 'C' type structure for each lattice site, all the discrete velocity values for a single lattice site are packed adjacent, and therefore lie consecutively in memory. The same holds for several cells of the grid, which lie consecutively in front and behind of each other.

Listing 3.1: Structure Cell : In 2D implementation.

```c
typedef struct
{
  /* 9 Lattice Velocities */
  double C;
  double E;
  double W;
  double N;
  double S;
  double T;
  double B;
  double NE;
  double NW;
  double SE;
  double SW;
} Cell;
```

Listing 3.2: Structure Cell : In 3D implementation.

```c
typedef struct
{
  /* 19 Lattice Velocities */
  double C;
  double E;
  double W;
  double N;
  double S;
  double T;
  double B;
  double NE;
  double NW;
  double SE;
  double SW;
  double TN;
  double TS;
  double TE;
  double TW;
  double BN;
  double BS;
  double BE;
  double BW;
} Cell;
```

In the implementation for this work, no additional ghost layers around the two grids are used, which were part in the previous LBM implementations. The ghost layer makes the implementations easy by checking only a single 'if' condition for all boundaries and obstacles inside the LBM kernel. Instead of this, the strategy was chosen to handle all boundary conditions and obstacles separately inside the LBM kernel. Alg. 3.1 shows the implementation of the six 'no-slip' boundary conditions for 3D case.

As in this thesis, mainly Klaus Iglberger's LBM kernel implementation has been followed, the basic implementation of the streaming and collide step of the Boltzmann equation is the same as in [4]. Additionally, all the previous optimization strategies are preserved in the initial implementation of the LBM kernel in 2D and 3D. The new ideas of optimization are discussed in chapter 4. The actual code can be found in App. B.

After implementing the LBM, Eq 1.7 and 1.8 results in the Alg. 3.2. Here all values from the 18 surrounding cells are gathered and written to the cell, which is currently updated and the complete cell update takes place. The 'if' condition on line 4 deals with the acceleration cells. The characteristic of an acceleration cell is, that here the velocity does not have to be computed, but rather is given from an external source. In this case, fixed values for the density rho and the 3 velocities ux, uy and uz are used. The 'else' part on line 9 is for the fluid cells. Here the values for the density and the velocities are computed. The lines 24 to 42 compute new values for the 19 velocities of the updated cell. This part corresponds to Eq. 1.7, which calculates new distribution functions from the equilibrium distribution and the old non-equilibrium distribution function.

3.4 Integration of COA and LBM

Matteo Frigo has published the 'cache oblivious algorithm' for any type of stencil based kernel computation. This makes LBM a valid candidate because the LBM is a stencil based calculation. For the D2Q9 model the LBM works with a 9 points stencil while for the D3Q19 model LBM works with a 19 points stencil. In each direction, the calculation depends on three consecutive points, which corresponds to a 3 point stencil calculation in each direction. In this thesis, three *Cache Oblivious Lattice Boltzmann Algorithms* (COLBA) were implemented.

1. A 2D COLBA using D2Q9 model.
2. A 3D COLBA using D3Q19 model.
3. An N-dimensional COLBA, which is flexible to adjust with any dimensional grid calculation.

3.4.1 2D and 3D Implementation of COLBA

The integration of COA and LBM to obtain a COLBA is straight forward. In the publication of Frigo [3], it is shown that any stencil based kernel can be added at a certain position of the algorithm to solve it in a cache oblivious manner. As the LBM is implemented as a LBM kernel this Kernel has only to be called at the appropriate location. Further more, the original one dimensional COA was extended for two and three dimensions respectively. For two and three dimensional case, an additional 'if' condition has been implemented. The idea is that the algorithm will first check in all dimensions if a space cut is necessary. If a space cut is not possible then it will do a time cut. This procedure recursively continues until the time

Algorithm 3.1 Lattice Boltzmann Kernel in 3D : boundary conditions

```
for $k = z0$ to $z1 - 1$ by 1 do
  for $i = x0$ to $x1 - 1$ by 1 do
    for $j = y0$ to $y1 - 1$ by 1 do
      if 0 == k then
        // Bottom Boundary
        dst(k,i,j).C = 0.0
        dst(k,i,j).T = src(k+1,i,j).B
        dst(k,i,j).TN = src(k+1,i+1,j).BS
        dst(k,i,j).TS = src(k+1,i-1,j).BN
        dst(k,i,j).TE = src(k+1,i,j+1).BW
        dst(k,i,j).TW = src(k+1,i,j-1).BE
      else if  LAYSIZE -1 == k then
        // Top Boundary
        dst(k,i,j).C = 0.0
        dst(k,i,j).B = src(k-1,i,j).T
        dst(k,i,j).BN = src(k-1,i+1,j).TS
        dst(k,i,j).BS = src(k-1,i-1,j).TN
        dst(k,i,j).BE = src(k-1,i,j+1).TW
        dst(k,i,j).BW = src(k-1,i,j-1).TE
      else if 0 == i then
        // South Boundary
        dst(k,i,j).C = 0.0
        dst(k,i,j).N = src(k,i+1,j).S
        dst(k,i,j).NE = src(k,i+1,j+1).SW
        dst(k,i,j).NW = src(k,i+1,j-1).SE
        dst(k,i,j).TN = src(k+1,i+1,j).BS
        dst(k,i,j).BN = src(k-1,i+1,j).TS
      else if ROWSIZE-1 == i then
        // North Boundary
        dst(k,i,j).C = 0.0
        dst(k,i,j).S = src(k,i-1,j).N
        dst(k,i,j).SE = src(k,i-1,j+1).NW
        dst(k,i,j).SW = src(k,i-1,j-1).NE
        dst(k,i,j).TS = src(k+1,i-1,j).BN
        dst(k,i,j).BS = src(k-1,i-1,j).TN
      else if 0 == j then
        // West Boundary
        dst(k,i,j).C = 0.0
        dst(k,i,j).E = src(k,i,j+1).W
        dst(k,i,j).SE = src(k,i-1,j+1).NW
        dst(k,i,j).NE = src(k,i+1,j+1).SW
        dst(k,i,j).TE = src(k+1,i,j+1).BW
        dst(k,i,j).BE = src(k-1,i,j+1).TW
      else if COLSIZE -1 == j then
        // East Boundary
        dst(k,i,j).C = 0.0
        dst(k,i,j).W = src(k,i,j-1).E
        dst(k,i,j).NW = src(k,i+1,j-1).SE
        dst(k,i,j).SW = src(k,i-1,j-1).NE
        dst(k,i,j).TW = src(k+1,i,j-1).BE
        dst(k,i,j).BW = src(k-1,i,j-1).TE
      else
        Solve fluid domain. Refer Alg. 3.2
      end if
    end for
  end for
end for
```

Algorithm 3.2 Lattice Boltzmann Kernel in 3D : acceleration and fluid Part

```
if Boundary then
    Check boundary conditions. Refer Alg. 3.1.
else
    if ACCELERATION && ROWSIZE-2 == i then
        ux = 0.1 * 3.0
        uy = 0.0
        uz = 0.0
        rho = 1.0
    else
        ux = src(k,i,j-1).E + src(k,i-1,j-1).NE + src(k,i+1,j-1).SE + src(k-1,i,j-1).TE + src(k+1,i,j-1).BE
        uy = src(k,i-1,j).N + src(k,i-1,j+1).NW + src(k-1,i-1,j).TN + src(k+1,i-1,j).BN
        uz = src(k-1,i,j).T + src(k-1,i+1,j).TS + src(k-1,i,j+1).TW
        rho = ux + uy + uz + src(k,i,j).C + src(k,i,j+1).W + src(k,i+1,j+1).SW + src(k,i+1,j).S + src(k+1,i,j).B + src(k+1,i+1,j).BS + src(k+1,i,j+1).BW
        rhoinv = 3/rho
        ux = (ux - src(k,i,j+1).W - src(k,i-1,j+1).NW - src(k,i+1,j+1).SW - src(k-1,i,j+1).TW - src(k+1,i,j+1).BW) * rhoinv
        uy = (uy + src(k,i-1,j-1).NE - src(k,i+1,j).S - src(k,i+1,j-1).SE - src(k,i+1,j+1).SW - src(k-1,i+1,j).TS - src(k+1,i+1,j).BS) * rhoinv
        uz = (uz + src(k-1,i-1,j).TN + src(k-1,i,j-1).TE - src(k+1,i,j).B - src(k+1,i-1,j).BN - src(k+1,i,j-1).BE - src(k+1,i+1,j).BS - src(k+1,i,j+1).BW) * rhoinv
    end if
end if
u_sqr = 1.0 - (0.1666666666666667 *( ux*ux + uy*uy + uz*uz))
w_0 = OMEGA * W_0 * rho
w_1 = OMEGA * W_1 * rho
w_2 = OMEGA * W_2 * rho
dst(k,i,j).C = (src(k,i,j).C * (1.0 - OMEGA)) + w_0 * (u_sqr)
dst(k,i,j).E = (src(k,i,j-1).E * (1.0 - OMEGA)) + w_1* ( u_sqr + ux + (0.5 * SQUAR(ux) ))
dst(k,i,j).W = (src(k,i,j+1).W * (1.0 - OMEGA)) + w_1* ( u_sqr - ux + (0.5 * SQUAR(ux) ))
dst(k,i,j).N = (src(k,i-1,j).N * (1.0 - OMEGA)) + w_1* ( u_sqr + uy + (0.5 * SQUAR(uy) ))
dst(k,i,j).S = (src(k,i+1,j).S * (1.0 - OMEGA)) + w_1* ( u_sqr - uy + (0.5 * SQUAR(uy) ))
dst(k,i,j).T = (src(k-1,i,j).T * (1.0 - OMEGA)) + w_1* ( u_sqr + uz + (0.5 * SQUAR(uz) ))
dst(k,i,j).B = (src(k+1,i,j).B * (1.0 - OMEGA)) + w_1* ( u_sqr - uz + (0.5 * SQUAR(uz) ))
dst(k,i,j).NE = src(k,i-1,j-1).NE * (1.0 - OMEGA) + w_2* ( u_sqr + (ux+uy) + (0.5 * SQUAR(ux+uy)))
dst(k,i,j).NW = src(k,i-1,j+1).NW * (1.0 - OMEGA) + w_2*( u_sqr + (-ux+uy) + (0.5 *SQUAR(-ux+uy)))
dst(k,i,j).SE = src(k,i+1,j-1).SE * (1.0 - OMEGA) + w_2* ( u_sqr + (ux-uy) + (0.5 * SQUAR(ux-uy)))
dst(k,i,j).SW = src(k,i+1,j+1).SW * (1.0 - OMEGA) + w_2*( u_sqr + (-ux-uy) + (0.5 *SQUAR(-ux-uy)))
dst(k,i,j).TN = src(k-1,i-1,j).TN * (1.0 - OMEGA) + w_2* ( u_sqr + (uz+uy) + (0.5 * SQUAR(uz+uy)))
dst(k,i,j).TS = src(k-1,i+1,j).TS * (1.0 - OMEGA) + w_2* ( u_sqr + (uz-uy) + (0.5 * SQUAR(uz-uy)))
dst(k,i,j).TW = src(k-1,i,j+1).TW * (1.0 - OMEGA) + w_2* ( u_sqr + (uz-ux) + (0.5 * SQUAR(uz-ux)))
dst(k,i,j).TE = src(k-1,i,j-1).TE * (1.0 - OMEGA) + w_2* ( u_sqr + (uz+ux) + (0.5 * SQUAR(uz+ux)))
dst(k,i,j).BN = src(k+1,i-1,j).BN * (1.0 - OMEGA) + w_2* ( u_sqr + (uy-uz) + (0.5 * SQUAR(uy-uz)))
dst(k,i,j).BS = src(k+1,i+1,j).BS * (1.0 - OMEGA) + w_2*( u_sqr + (-uy-uz) + (0.5 *SQUAR(-uy-uz)))
dst(k,i,j).BW = src(k+1,i,j+1).BW * (1.0 - OMEGA) + w_2*( u_sqr + (-ux-uz) + (0.5 *SQUAR(-ux-uz)))
dst(k,i,j).BE = src(k+1,i,j-1).BE * (1.0 - OMEGA) + w_2* ( u_sqr + (ux-uz) + (0.5 * SQUAR(ux-uz)))
```

difference *dt* is 1. Alg. 3.3 and Alg. 3.4 show the implementation of COLBA. Line number 7, 9, 14 and 16 indicates the function call of the LBM kernel. The traversal of COLBA for 1D space and 1D time domain is shown in Fig. 3.5. In 2D and 3D case, the COLBA behaves in similar way along other two directions. Vertical levels in each diagram of Fig. 3.5 represent time scale. Cells in light(yellow) color represent the updated cells for that particular time step while dark(green) color cells represent old updates. The first diagram shows update of 6 cells for first time step. The second and third diagram shows update of 3 and 2 cells for second and third time steps respectively. The COLBA proceeds in the same way, diagrams 4, 5 and 6 show update of successive cells for second, third and first time steps respectively. The number of cells updated in each stage depend on the space cut, number of time steps and domain size.

Algorithm 3.3 COLBA : 2D single processor implementation

```
 1: void walk2D(int t0, int t1, int x0, int dx0, int x1, int dx1, int y0, int dy0, int y1, int dy1)
 2: int dt = t1 - t0
 3: if dt == 1 then
 4:   if (t0 & 1) != 0 then
 5:     // Odd time step
 6:     if (0 == x0 || 0 == y0 || ROWSIZE == x1 || COLSIZE == y1) then
 7:       LBMKernel3D( dst3D, src3D, x0, x1, y0, y1)
 8:     else
 9:       LBMKernel3D_withoutBnd( dst3D, src3D, x0, x1, y0, y1)
10:     end if
11:   else
12:     // Even time step
13:     if (0 == x0 || 0 == y0 || ROWSIZE == x1 || COLSIZE == y1)  then
14:       LBMKernel3D( src3D, dst3D, x0, x1, y0, y1)
15:     else
16:       LBMKernel3D_withoutBnd( src3D, dst3D, x0, x1, y0, y1)
17:     end if
18:   end if
19: else
20:   if (SPC * (x1 - x0) + (dx1 - dx0) * dt $\geq$ 4 * dt) then
21:     // Space cut in X direction
22:     int xm = (int)(( 2*(x1+x0) + ((2+dx0+dx1)*dt) )$\gg$2)
23:     walk2D(t0, t1, x0, dx0, xm, -1, y0, dy0, y1, dy1)
24:     walk2D(t0, t1, xm, -1, x1, dx1, y0, dy0, y1, dy1)
25:   else if (SPC * (y1 - y0) + (dy1 - dy0) * dt geq 4 * dt) then
26:     // Space cut in Y direction
27:     int ym = (int)((2*(y0+y1) + ((2+dy0+dy1)*dt))$\gg$2)
28:     walk2D(t0, t1, x0, dx0, x1, dx1, y0, dy0, ym, -1)
29:     walk2D(t0, t1, x0, dx0, x1, dx1, ym, -1, y1, dy1)
30:   else
31:     // Time cut
32:     int s = dt/2
33:     walk2D( t0, t0+s, x0, dx0, x1, dx1, y0, dy0, y1, dy1)
34:     walk2D( t0+s, t1, (x0+(dx0*s)), dx0, (x1+(dx1*s)), dx1, (y0+(dy0*s)), dy0, (y1+(dy1*s)), dy1)
35:   end if
36: end if
```

Algorithm 3.4 COLBA : 3D single processor implementation

```
void walk3D(int t0, int t1, int x0, int dx0, int x1, int dx1, int y0, int dy0, int y1, int dy1, int z0, int dz0, int z1, int dz1)
int dt = t1 - t0
if dt == 1 then
  if (t0 & 1) != 0 then
    // Odd time step
    if (0 == x0 || 0 == y0 || 0 == z0 || ROWSIZE == x1 || COLSIZE == y1 || LAYSIZE == z1) then
      LBMKernel3D( dst3D, src3D, x0, x1, y0, y1, z0, z1)
    else
      LBMKernel3D_withoutBnd( dst3D, src3D, x0, x1, y0, y1, z0, z1)
    end if
  else
    // Even time step
    if (0 == x0 || 0 == y0 || 0 == z0 || ROWSIZE == x1 || COLSIZE == y1 || LAYSIZE == z1) then
      LBMKernel3D( src3D, dst3D, x0, x1, y0, y1, z0, z1)
    else
      LBMKernel3D_withoutBnd( src3D, dst3D, x0, x1, y0, y1, z0, z1)
    end if
  end if
else
  if (SPC * (x1 - x0) + (dx1 - dx0) * dt $\geq$ 4 * dt) then
    int xm = (int)(( 2*(x1+x0) + ((2+dx0+dx1)*dt) ) $\gg$ 2)
    walk3D(t0, t1, x0, dx0, xm, -1, y0, dy0, y1, dy1, z0, dz0, z1, dz1)
    walk3D(t0, t1, xm, -1, x1, dx1, y0, dy0, y1, dy1, z0, dz0, z1, dz1)
  else if (SPC * (y1 - y0) + (dy1 - dy0) * dt geq 4 * dt) then
    int ym = (int)((2*(y0+y1) + ((2+dy0+dy1)*dt)) $\gg$ 2)
    walk3D(t0, t1, x0, dx0, x1, dx1, y0, dy0, ym, -1, z0, dz0, z1, dz1)
    walk3D(t0, t1, x0, dx0, x1, dx1, ym, -1, y1, dy1, z0, dz0, z1, dz1)
  else if (SPC * (z1 - z0) + (dz1 - dz0) * dt geq 4 * dt ) then
    int zm = (int)((2*(z0+z1) + ((2+dz0+dz1)*dt))$\gg$ 2)
    walk3D(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1, z0, dz0, zm, -1)
    walk3D(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1, zm, -1, z1, dz1)
  else
    int s = dt/2
    walk3D( t0, t0+s, x0, dx0, x1, dx1, y0, dy0, y1, dy1, z0, dz0, z1, dz1)
    walk3D( t0+s, t1, (x0+(dx0*s)), dx0, (x1+(dx1*s)), dx1, (y0+(dy0*s)), dy0, (y1+(dy1*s)), dy1, (z0+(dz0*s)), dz0, (z1+(dz1*s)), dz1)
  end if
end if
```

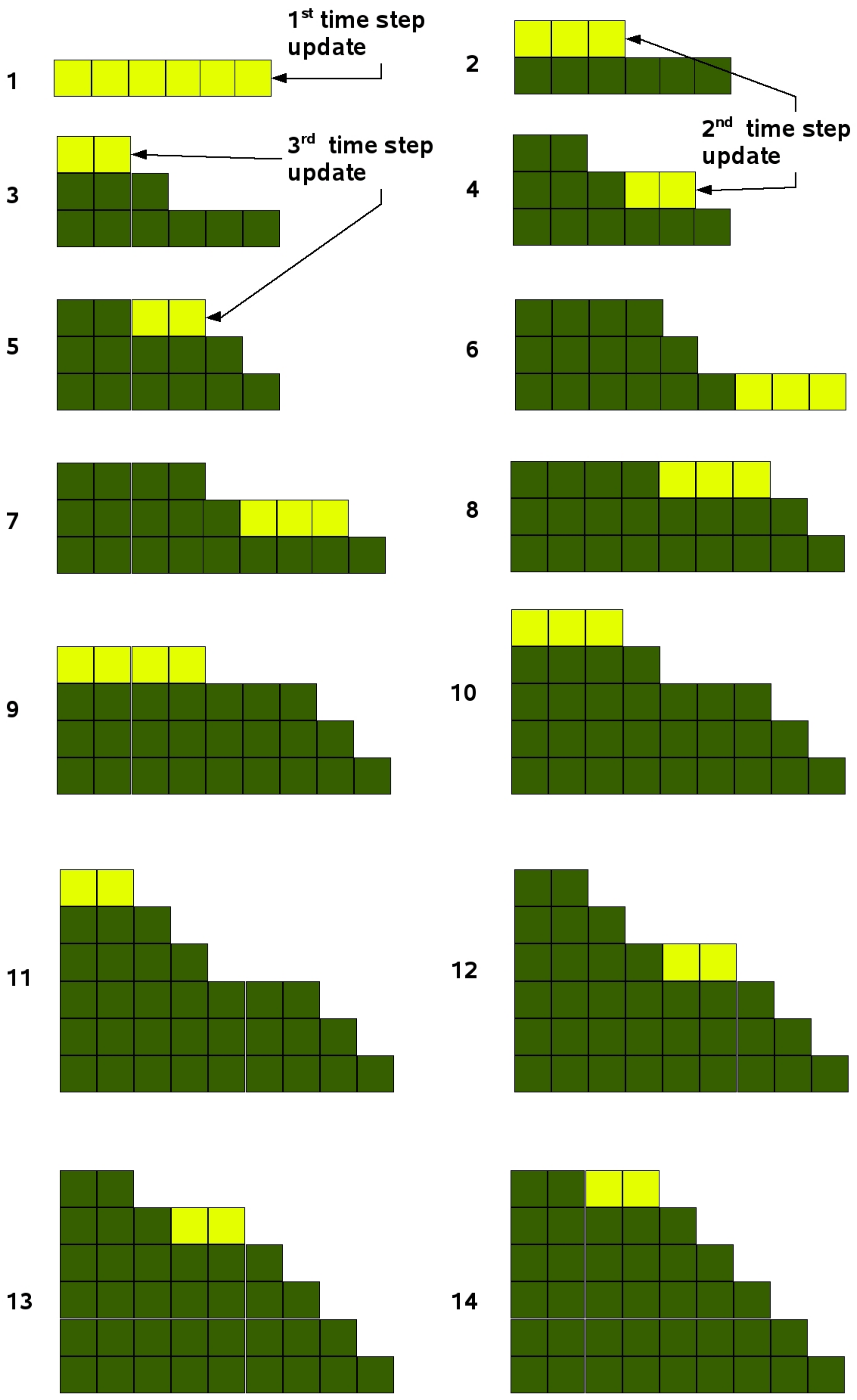


Figure 3.5: Step by step illustration of COLBA traversal in 1D space and 1D time domain. The cell update scheme depends on space cut factor, domain size and number of time steps.

3.4.2 Multi (N-) Dimensional Implementation of COLBA

In the multi-dimensional implementation, the procedure is generalized for an arbitrary-dimensional space-cut. Here, a N-dimensional stencil is considered , where $N > 0$ is the number of space dimension (excluding time). For each dimension, the projection of the multi-dimensional trapezoid on the plane looks like the one dimensional trapezoid as shown in Fig. 2.1. Consequently, we can apply the same recursive decomposition that we used in the one dimensional case. If any dimension permits a space cut in its plane, then a space cut is performed in that dimension. Otherwise, if none of the space dimensions can be split, a time cut is applied in the same fashion as in the one dimensional case.

Alg. 3.5 shows the multi-dimensional implementation of COLBA. Each trapezoid in one dimension is represented by a C type structure. The function *walkND()* takes parameters as a structure pointer, the start time step and the end time step. The structure pointer actually points to an array of structure 'Dimension' in which each array element represents one dimension. The size of the array is equal to the number of dimensions.

The *for* loop inside the walkND function runs over dimension array and sequentially checks each entry for the space-cut condition. If any dimension permits a space-cut, then the space-cut is done in that dimension, otherwise a time-cut is applied. Once a space-cut or time-cut is done walkND, calls itself recursively on two newly created trapezoids or parallelograms. The procedure continues until *dt* is one, in which case the LBM kernel is solved.

3.5 Model Geometry

This work has focused on simple geometries to prove the adaptability of the cache-oblivious blocking approach to the LBM. Therefore, the code is mainly tested for the *Lid-Driven Cavity problem*, which describes the motion of fluid in a box, where the top wall is endlessly sliding into the same direction at constant speed. This is simulated by assigning the cells of the first layer a constant speed in x-direction. For the remaining three walls, no-slip condition are applied. The flow field was visualized with the help of OpenDX [1] for both the 2D and 3D cases. The result is shown in Fig. 3.6. Even though the testing is done for simple geometry, the implementation is flexible and can be adapted to complex geometries.

Algorithm 3.5 COLBA: ND single processor implementation

```
void walkND(int t0, int t1, Dimention* dim, int n)
Dimention *d
int dt = t1 - t0
int cnt = 0
if dt == 1 then
    if (t0 & 1) != 0 then
        if (0 == dim[1].x0 || 0 == dim[2].x0 || 0 == dim[0].x0 || ROWSIZE == dim[1].x1 || COLSIZE ==
        dim[2].x1 || LAYSIZE == dim[0].x1) then
            LBMKernel3D( dst3D, src3D, dim[1].x0, dim[1].x1, dim[2].x0, dim[2].x1, dim[0].x0, dim[0].x1)
        else
            LBMKernel3D_withoutBnd( dst3D, src3D, dim[1].x0, dim[1].x1, dim[2].x0, dim[2].x1, dim[0].x0,
            dim[0].x1)
        end if
    else
        if (0 == dim[1].x0 || 0 == dim[2].x0 || 0 == dim[0].x0 || ROWSIZE == dim[1].x1 || COLSIZE ==
        dim[2].x1 || LAYSIZE == dim[0].x1) then
            LBMKernel3D( src3D, dst3D, dim[1].x0, dim[1].x1, dim[2].x0, dim[2].x1, dim[0].x0, dim[0].x1)
        else
            LBMKernel3D_withoutBnd( src3D, dst3D, dim[1].x0, dim[1].x1, dim[2].x0, dim[2].x1, dim[0].x0,
            dim[0].x1)
        end if
    end if
else
    for $d = dim$ to $dim + n$ by 1 do
        int x0 = d → x0
        int dx0 = d → dx0
        int x1 = d → x1
        int dx1 = d → dx1
        if SPC * (x1 - x0) + (dx1 - dx0) * dt ≥ 4 * dt then
            Dimention save = *d
            int xm = (int)(( 2*(x1+x0) + ((2+dx0+dx1)*dt) ) ≫ 2)
            *d = (Dimention)x0, dx0, xm, -1
            walkND(t0, t1, dim, n)
            *d = (Dimention)xm, -1, x1, dx1
            walkND(t0, t1, dim, n)
            *d = save
            return
        end if
    end for
    int s = (dt/2)
    Dimention newDim[n]
    int i = 0
    walkND(t0, t0+s, dim, n)
    for ($i = 0$ to $n - 1$ by 1 do
        newDim[i] = (Dimention)dim[i].x0 + dim[i].dx0 * s, dim[i].dx0, dim[i].x1 + dim[i].dx1 * s, dim[i].dx1
    end for
    walkND(t0+s, t1, newDim, n)
end if
```

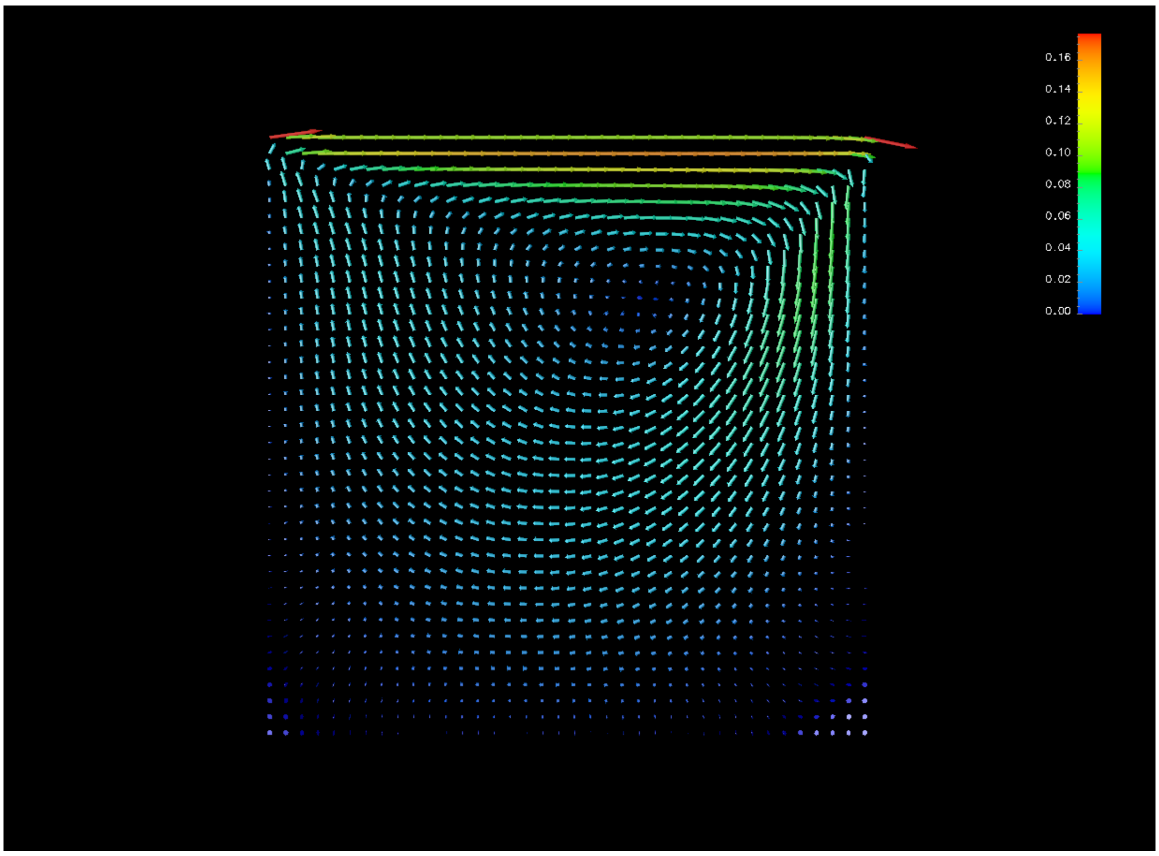


Figure 3.6: Visualization of the lid-driven cavity problem with OpenDX (domain size = 40^3 ,time steps = 10000, omega = 1.5)

Chapter 4

Optimization Strategies

Before discussing the various optimization strategies, the focus of this work should be clarified. The following points directly or indirectly affect the optimization strategies.

1. **Objective of the project:** The main objective of the project was to implement an efficient LBM using the new stencil based COA. It has the inherent characteristic of reducing the cache misses. Therefore it was to be expected that the performance will improve significantly compared to an unoptimized LBM implementation. Our notion is reasonable because cache misses are one of the major hurdles in performance improvement.

2. The aim was to outperform a blocked version or at least raise the performance of the COLBA to the performance of a blocked version.

3. The optimization strategies for 2D and 3D are the same. After achieving convincing results for the 2D case, the 3D case was examined. So in the case of 3D, the optimizations applied in 2D were present from the beginning. Each new optimization is progressively combined with the previous optimization.

4. As performance measuring unit, MLUps (Mega Lattice Update site per second) is used, which is calculated with the following formula:

$$\frac{\text{x-size * y-size * z-size * No. of time steps}}{10^6\text{* time taken}}$$

5. All ideas or strategies applied for the optimization are discussed. Some of these strategies worked, some did not.

Even though all optimization strategies from previous work were preserved, the initial performance of COLBA was not outstanding. It showed hardly better performance than the iterative version. The positive aspect was that the algorithm behaved according to our expectation: it showed a substantial reduction in the number of cache misses. Fig. 4.1 shows the result from the initial run of COLBA.

The resulting question was where the algorithm was consuming the extra time gained from the reduced number of cache misses. The immediate target was the LBM kernel and the

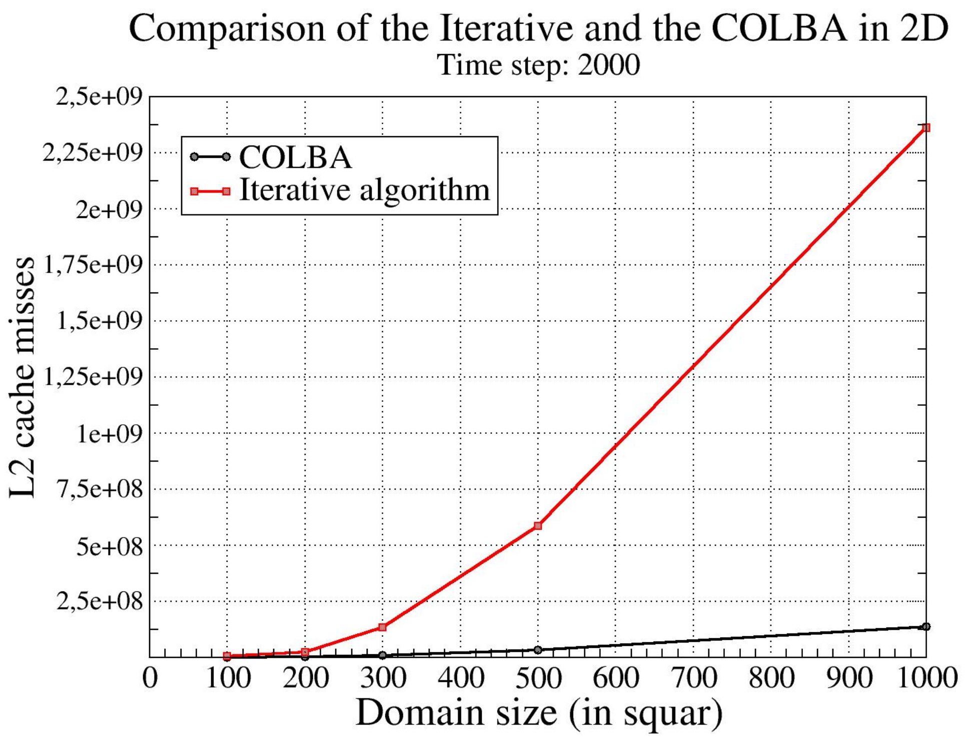


Figure 4.1: Initial graph showing L2 cache misses for a varying domain size. The measurements were done with PAPI on fauia59. App. A.

recursive function call overhead. The function call overhead has been checked by commenting out the LBM kernel function call. It showed to be negligible (1-2 % of the total time). This result shifted the focus on the LBM kernel, which consists of more than 250 FLOPs [4] and 5 (in case of 2D) or 7 (in case of 3D) *if*-conditions. This favored the decision to opt for more compiler based optimization options and to emphasize on the internal optimization of the LBM kernel i.e. code optimizations.

4.1 Implemented Strategies

4.1.1 Compiler Flags for Optimization

Compiler based optimization is done by using compiler flags. The various compiler flags used for the optimization are discussed below. Every optimization option has its own contribution in optimizing the algorithm, and each single option improved the performance. However, the individual contributions of each compiler option were not measured.

1. **-O3:** Does aggressive optimization speed which may increase the compilation time. Turns on all optimization specified by -O2. (for details refer the ICC or the GCC manual.)

2. **-xW, -xP:** It generates specialized processor specific code. Both the options are Intel Pentium 4 specific and compatible with other Intel processors. These options are IA-32 architecture specific and also available for Intel EM64T (IA-64) processor. These flags can not be used for the Itanium processor family. As these options are available only for the Intel compiler, they can not be used with the gcc compiler.

3. **-fno-alias:** Assumes no aliasing in the program. The default setting is 'off'.

4. **-prof_gen and -prof_use:**These two options are used for a profile guided optimization. During the first run of the program, *-prof_gen* instructs the compiler to produce instrumented code. The successive run of the program with *prof_use* uses the previously gathered information as a guideline to produce highly optimized code. This option works best for code with many frequently executed branches that are difficult to predict at compile time. Since COLBA is a recursive algorithm, it is hard to do a branch prediction at compile time. Unfortunately, this options did not increase the performance of COLBA. One reason for this could be, that it is difficult to maintain all information about all function calls of recursive functions.

The detailed documentation for the above compiler flags can be found on the manual pages of gcc and icc compiler.

4.1.2 Switch Over to ICC from GCC Compiler

The compiler plays a significant role to compile the program efficiently. Switching from the GNU compiler to the Intel compiler brought a significant performance improvement. As stated above the ICC provides some additional optimization flags, which are not available with the GCC. Fig. 4.2 shows the performance of COLBA, using the GCC and the ICC compiler.

4.1.3 Different Space Cut (SPC) Factors

In the COLBA, there is a condition for cutting space-time domain. Initially, the derivation of a mathematical base for this condition was attempted, but the introduction of the space-cut factor (SPC) proved to be sufficient. The choice of the SPC is describe below: If the space domain is twice as large as the time domain then the space domain is cut, else the time domain is cut. This process recursively proceeds until the size of the time domain becomes one.

Basically, the SPC determines the size of the domain for which the LBM kernel is solved. Two consequences can occur if the SPC is increased or decreased. If the SPC is gradually decreased, then the final domain size starts increasing and at some point the problem may not be able to be solved within the cache. If the SPC is gradually increased, then at some point the problem size becomes equal to one cell, which results in a bad performance. Tab. 4.1 shows the approximate patch sizes for different SPCs.

The influence of the SPC was studied by varying the factor in the range of 0.25 to 4. In the case of 2D and 3D, instead of having the same SPC in all direction, different combinations of SPCs can be applied to the different directions. This was another optimization approach,

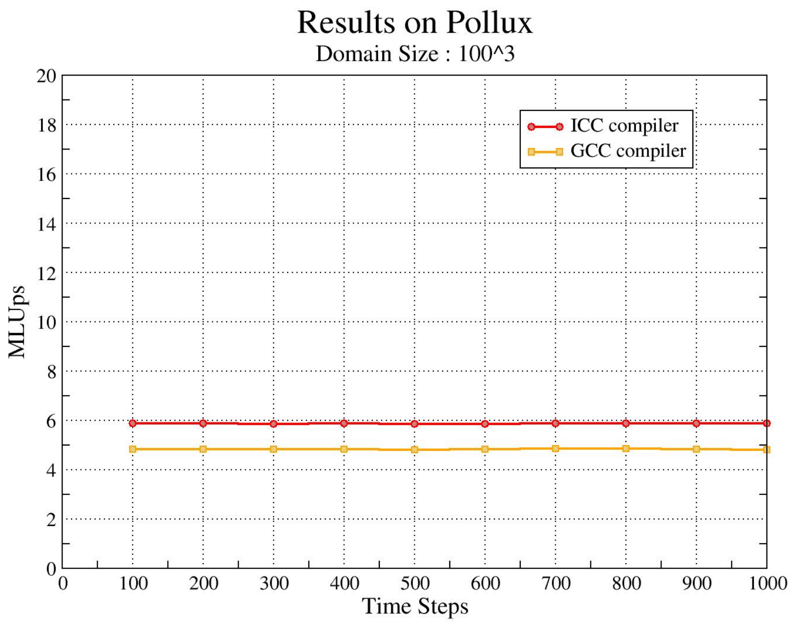


Figure 4.2: Comparison of performance between the icc and the gcc compiled version on Pollux system.

	Doman Size	
SPC	100	10
0.25	26	10
2.0	3	4
3.5	2	2

Table 4.1: Table showing approximate solvable patch sizes created by COLBA for 10 number of time steps.

but it too did not show any improvement in performance. The best performance has been observed was for a SPC of value 2 for both the 2D and 3D case. Tab. 4.2 shows the comparison of the different SPCs.

4.1.4 Splitting of LBM Kernel into 2 Functions

The primary reason for the COLBA being slow was the inefficient LBM kernel. This can be seen by skipping the actual LBM calculations in COLBA. In this case it could be monitored that the overhead of COA is negligible. Key to an efficient LBM kernel is the elimination of the boundary checks in the LBM loops (inside LBM kernel). Therefore the LBM kernel is implemented in two different functions: the first one will only solve the fluid part without boundary checks and the second one with boundary checks. The COLBA always works with small patches. Normally, the final solvable patch size varies in the range 2^2 - 4^2 in 2D and 2^3

SPC	MLUPs (Pollux)	MLUPs (Woodcrest)	MLUp (sfront03)
0.25	4.81	6.39	–
0.5	5.60	8.19	–
1.0	6.48	9.86	5.6
1.5	6.68	–	5.75
2.0	6.57	10.1	5.85
2.5	6.55	10.21	–
3.0	6.21	9.76	–
3.5	6.04	–	–

Table 4.2: Analysis of SPC. Results for 100^3 domain size and 100 time steps.

- 4^3 in 3D. Bigger patches are always reduced to this size by COLBA. These small patches can be differentiated as patches with boundary cells and patches completely surrounded by fluid cells. In a normal domain size of 100^3, 95% of the time pure fluid patches are solved. For such patches there is no need to check boundary conditions. Additionally, as the domain size increases, the percentage of fluid patches increases as well. Fig. 4.3 illustrates this circumstance. Both functions are listed in App. B.

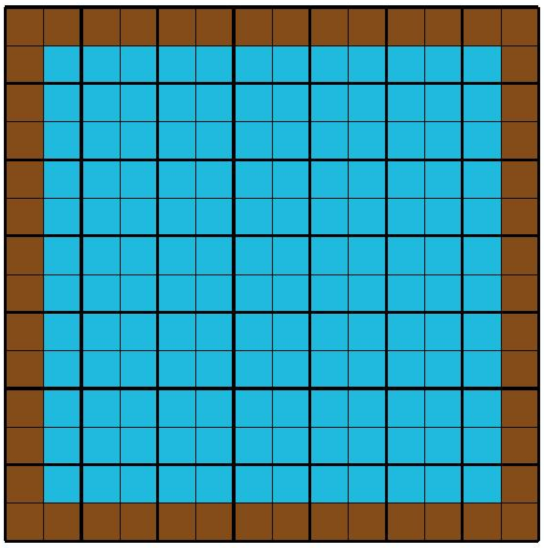


Figure 4.3: Diagram showing fluid patches with and without boundary cells. The inner cells are fluid cells and the outer cells represent boundary cells.

4.1.5 Usage of the Compiler Directives

The usage of *#Pragma ivdep* and *#Pragma vector always* proved to be very effective when placed at the beginning of a *for* loop. *#pragma ivdep* indicates the compiler that it should ignore the vector dependences, i.e. there is absolutely no loop-carried memory dependency in the loop where this pragma is placed. *#pragma vector always* will vectorize non-unit strides or very unaligned memory accesses. Both pragmas help the compiler to vectorize the code. These pragmas fail if more than one *if* condition is present inside the *for* loop'. Therefore, only in the function LBM kernel without boundary these pragmas turned out to be useful. Alg. 4.1 shows the placement of the pragmas.

Algorithm 4.1 LBM in 3D : Usage of #pragma ivdep and vector always and rearrangement of the ACCELERATION *if* condition.

```
void LBMKernel3D_withoutBnd(Cell* src, Cell* dst, int x0, int x1, int y0, int y1, int z0, int z1)
for $k = z0$ to $z1 - 1$ by 1 do
  for $i = x0$ to $x1 - 1$ by 1 do
    if ACCELERATION && ROWSIZE-2 == i then
      ux = 0.1 * 3.0
      uy = 0.0
      uz = 0.0
      rho = 1.0
      u_sqr = 1.0 - (0.1666666666666667 *( ux*ux + uy*uy + uz*uz))
        ⋮                    ⋮
      # pragma ivdep
      # pragma vector always
      for $j = y0$ to $y1 - 1$ by 1 do
        // Calculate 19 distribution functions for each cell.
      end for
    else
      # pragma ivdep
      # pragma vector always
      for $j = y0$ to $y1 - 1$ by 1 do
        ux = ...
        uy = ...
        uz = ...
          ⋮                    ⋮
        u_sqr = 1.0 - (0.1666666666666667 *( ux*ux + uy*uy + uz*uz))
          ⋮                    ⋮
        // Calculate 19 distribution functions for each cell.
      end for
    end if
  end for
end for
```

4.1.6 Rearrangement of Instructions to Reduce Floating Point Operations

Splitting the *LBMKernel* function in two parts (with and without boundary), It has been observed that the *if* condition for the ACCELERATION cells could be moved out of the innermost j-loop and some floating point operations could be saved. Moving the *if* condition outside the *for* loop helped in vectorizing the *for* loop. Alg 4.1 shows the arrangement of the *if* condition and the *for* loop while Alg 4.2 shows the changes in the floating point operations: the 'gray' color text shows the old code, 'bold' text shows changed code and normal text show unchanged code. These changes helped to improve the performance approximately by 5-7%.

Algorithm 4.2 Lattice Boltzmann Kernel in 3D : Rearrangement of floating point operations.

```
for $k = z0$ to $z1 - 1$ by 1 do
  for $i = x0$ to $x1 - 1$ by 1 do
    for $j = y0$ to $y1 - 1$ by 1 do
      if Boundary then
        check boundary conditions
      else
        if ACCELERATION && ROWSIZE-2 == i then
          ux = 0.1
          ux = 0.1 * 3.0
          ⋮        ⋮
        else
          ux = src(k,i,j-1).E + src(k,i-1,j-1).NE + src(k,i+1,j-1).SE + src(k-1,i,j-1).TE + src(k+1,i,j-1).BE
          uy = src(k,i-1,j).N + src(k,i-1,j+1).NW + src(k-1,i-1,j).TN + src(k+1,i-1,j).BN
          uz = src(k-1,i,j).T + src(k-1,i+1,j).TS + src(k-1,i,j+1).TW
          rho = ux + uy + uz + src(k,i,j).C + src(k,i,j+1).W + src(k,i+1,j+1).SW + src(k,i+1,j).S + src(k+1,i,j).B + src(k+1,i+1,j).BS + src(k+1,i,j+1).BW
          rhoinv = 1/rho
          rhoinv = 3/rho
          ux = (ux - src(k,i,j+1).W - src(k,i-1,j+1).NW - src(k,i+1,j+1).SW - src(k-1,i,j+1).TW - src(k+1,i,j+1).BW) * rhoinv
          uy = ...
          uz = ...
        end if
      end if
      u_sqr = (1.5 *( ux*ux + uy*uy + uz*uz))
      u_sqr = 1.0 - (0.1666666666666667 *( ux*ux + uy*uy + uz*uz))
        ⋮            ⋮
      dst(k,i,j).C = (src(k,i,j).C * (1.0 - OMEGA)) + w_0 * (u_sqr)
      dst(k,i,j).E = (src(k,i,j-1).E * (1.0 - OMEGA)) + w_1* ( 1-u_sqr + 3.0*ux + (4.5 * SQUAR(ux) ))
      dst(k,i,j).E = (src(k,i,j-1).E * (1.0 - OMEGA)) + w_1* ( u_sqr + ux + (0.5 * SQUAR(ux) ))
        ⋮            ⋮            ⋮
        ⋮            ⋮            ⋮
      dst(k,i,j).SW = src(k,i+1,j+1).SW * (1.0 - OMEGA) + w_2*( 1-u_sqr + 3.0*(-ux-uy) + (4.5 *SQUAR(-ux-uy)))
      dst(k,i,j).SW = src(k,i+1,j+1).SW * (1.0 - OMEGA) + w_2*( u_sqr - (ux+uy) + (0.5 *SQUAR(ux+uy)))
        ⋮            ⋮            ⋮
        ⋮            ⋮            ⋮
      dst(k,i,j).BW = src(k+1,i,j+1).BW * (1.0 - OMEGA) + w_2*( 1-u_sqr + 3.0*(-ux-uz) + (4.5 *SQUAR(-ux-uz)))
      dst(k,i,j).BW = src(k+1,i,j+1).BW * (1.0 - OMEGA) + w_2*( u_sqr - (uz+ux) + (0.5 *SQUAR(ux+uz)))
        ⋮            ⋮            ⋮
        ⋮            ⋮            ⋮
    end for
  end for
end for
```

The above three strategies, i.e. splitting the LBM kernel into two functions, usage of #pragma ivdep and #pragma vector always and the rearrangement of instructions were the key for a fast LBM kernel implementation.

4.1.7 Reduction of the Function Parameters

Through the analyzation of the COLBA with Intel's VTune, a register spill on large scale has been observed. The reason for this additional overhead is the large number of function parameters for the walk3D() function, which is called recursively. From the 16 parameters of the 'walk3D()' function, only 2 parameters could be reduced, i.e. the source and destination arrays could be declared as global variables. For the other parameters, data compression was tried . i.e. while passing the parameters to the function, 2 *int* variables are packed in 1 *long_long* variable and then unpacked in the function. This did not result in a performance gain as the data uncompression costs registers equally as in the previous case. However, the register spills were not the primary reason for the COLBA staying behind our expectations. This was concluded from another VTune analyzation with a commented *LBMKernel()* function. Even though the number of register spills was the same, the overhead was negligible. For this reason focus shifted on the optimization of the LBM kernel.

4.1.8 Prefetching Small Patches Encountered by COLBA

From the beginning, the reference for the performance measurements of COLBA was the performance of Stefan Donath's LBM kernel, which was developed with an optimal data structure and which is the fastest among all existing LBM kernels at the RRZE. When both the LBM kernels were profiled with 'VTune' and 'HPCMon', it was observed that the COLBA prefetches less data compared to the faster LBM kernel. The analysis and graphs of the 'VTune' profiling and the 'HPCMon' results are mentioned in Chapter 5. As already described in previous chapter, the COLBA recursively divides the complete space-time domain in small patches and then solves these patches one by one. The idea to increase prefetching was to prefetch the $(n+1)^{th}$ patch before solving the n^{th} patch. This could minimize the impact of memory latency.

1. **Index table approach**

   This approach aims at creating an index table, which stores the information of all patches created by the COLBA. For the implementation, an array *indexTable* of structure 'Indices' was allocated which is of equal length to the number of small patches created by the COLBA. This information is gathered by running the COLBA without the LBM kernel. The disadvantage of this approach is that a huge index table was created, which takes a substantial amount of memory and creates unnecessary traffic in the cache. Therefore, instead of using the index table, prefetching inside the COLBA was attempted.

2. **Prefetching inside COLBA**

   The implementation is shown in Alg 4.3. Here, only change compared to the original COLBA is an additional call for prefetching the next block. The bold text indicates the additional prefetching code. For prefetching the next patch, *_mm_prefetch()* subroutine was used. It loads data from the main memory to a location "closer" to the processor (L2 cache) before it is needed. In Alg. 4.4 the call of this subroutine is shown. For more details see [5].

Listing 4.1: Prefetching : An index table approach. Structure to store one patch information.

```
typedef struct
{
  int t0;
  int t1;
  int x0;
  int x1;
  int y0;
  int y1;
  int z0;
  int z1;
} Indices;

Indices* indexTable;
```

It can be seen from the profiling results on HPCMon that the second prefetching strategy reduced the prefetch bus access and increased the prefetching but the prefetching did not show any improvement in performance.

4.1.9 Solving COLBA for Small Chunks of Time Steps over Complete Space Domain

As the COLBA operates on small patches, one particular patch is for example solved for time step t, where as other patches are solved for time steps $t+1$, $t+2$ or $t+3$. This may cause cache misses because different sections of the arrays are accessed during the calculation.

In Alg. 4.5, the variable *timeFact* decides the size of the time steps so that in one iteration of the *for* loop *timeFact* time steps are solved. The *for* loop runs over all chunks. This strategy too did not show any improvement.

In addition to the above listed schemes, consideration was given to two other schemes, "implementation using partially iterative and recursive algorithm" and "LBM kernel macro". But after getting the major break through mentioned above, COLBA (recursive version) started running 1.5 - 2 times faster than iterative version. So there was no point in combining slower algorithm with faster code. The LBM kernel macro was tested for 2D case, but it also did not show any improvement in performance. As the LBM kernel is very heavy in terms of floating point operations, the function call overhead is negligible in front of its core computation. Therefore, both the schemes have been discarded.

Algorithm 4.3 Prefetching inside COLBA

```
 1: void walk3D_withPrefetch(int t0, int t1, int x0, int dx0, int x1, int dx1, int y0, int dy0, int y1, int dy1,
    int z0, int dz0, int z1, int dz1)
 2: int dt = t1 - t0;
 3: if dt == 1 then
 4:   if First patch then
 5:     prefetch the first patch. // See App. B for detail code.
 6:   else
 7:     Solve $(n-1)^{th}$ patch.
 8:     // prefetch $n^{th}$ patch.
 9:     prefetchNextBlock(x0, x1, y0, y1, z0, z1)
10:   end if
11: else
12:   if (SPC * (x1 - x0) + (dx1 - dx0) * dt $\geq$ 4 * dt) then
13:     int xm = (int)(( 2*(x1+x0) + ((2+dx0+dx1)*dt) )$\gg$2)
14:     walk3D_withPrefetch(t0, t1, x0, dx0, xm, -1, y0, dy0, y1, dy1, z0, dz0, z1, dz1)
15:     walk3D_withPrefetch(t0, t1, xm, -1, x1, dx1, y0, dy0, y1, dy1, z0, dz0, z1, dz1)
16:   else if (SPC * (y1 - y0) + (dy1 - dy0) * dt $\geq$ 4 * dt) then
17:     int ym = (int)((2*(y0+y1) + ((2+dy0+dy1)*dt))$\gg$2)
18:     walk3D_withPrefetch(t0, t1, x0, dx0, x1, dx1, y0, dy0, ym, -1, z0, dz0, z1, dz1)
19:     walk3D_withPrefetch(t0, t1, x0, dx0, x1, dx1, ym, -1, y1, dy1, z0, dz0, z1, dz1)
20:   else if (SPC * (z1 - z0) + (dz1 - dz0) * dt $\geq$ 4 * dt ) then
21:     int zm = (int)((2*(z0+z1) + ((2+dz0+dz1)*dt))$\gg$2)
22:     walk3D_withPrefetch(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1, z0, dz0, zm, -1)
23:     walk3D_withPrefetch(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1, zm, -1, z1, dz1)
24:   else
25:     int s = dt/2
26:     walk3D_withPrefetch( t0, t0+s, x0, dx0, x1, dx1, y0, dy0, y1, dy1, z0, dz0, z1, dz1)
27:     walk3D_withPrefetch( t0+s, t1, (x0+(dx0*s)), dx0, (x1+(dx1*s)), dx1, (y0+(dy0*s)), dy0,
        (y1+(dy1*s)), dy1, (z0+(dz0*s)), dz0, (z1+(dz1*s)), dz1)
28:   end if
29: end if
```

Algorithm 4.4 Function for prefetching next block in cache

```
1: void prefetchNextBlock(int x0, int x1, int y0, int y1, int z0, int z1)
2: for $k = z0$ to $z1 - 1$ by 1 do
3:   for $i = x0$ to $x1 - 1$ by 1 do
4:     for $j = y0$ to $y1 - 1$ by 1) do
5:       _mm_prefetch((char*) &src3D(k,i,j).C, _MM_HINT_T0)
6:       _mm_prefetch((char*) &dst3D(k,i,j).C, _MM_HINT_T0)
7:     end for
8:   end for
9: end for
```

Algorithm 4.5 Optimization Strategy : Solving complete domain for small chunk of time steps

```
1: void callFrigoMultiple(int t0, int t1, int x0, int dx0, int x1, int dx1, int y0, int dy0, int y1, int dy1, int
   z0, int dz0, int z1, int dz1)
2: int timeFact = 10
3: for $i = 0$ to $(t1/timeFact) - 1$ by 1 do
4:   walk3D(t0, timeFact, x0, dx0, x1, dx1, y0, dy0, y1, dy1, z0, dz0, z1, dz1)
5: end for
```

4.2 Profiling Tools:

1. **PAPI:** PAPI stands for Performance Application Programming Interface. Hardware performance counters are available on most modern microprocessors. PAPI provides interface to these underlying hardware counters through events. Thus it provides a number of events to capture various details of the program such as L1, L2, L3 data and instruction cache misses or hits, floating point operations, load and store misses etc. Instrumentation of the code is required. i.e. first initialize PAPI library and then call appropriate event at appropriate location inside the code. It also provides portability across different platforms. It uses the same routines with similar argument lists to control and access the counters for every architecture.

2. **Valgrind and KCachegrind:** Valgrind is a free profiling tools from GNU for programs running under Linux. It is a simulation-based profiling tools which consists of a core, which provides a synthetic CPU in software. It offers a number of tools for program analysis such as memcheck, cachegrind, callgrind etc. and works directly with the existing executables. There is no need to recompile, relink, or modify, the program.

   KCachegrind is a visualization tool for the profiling data generated by cachegrind. The cachegrind is a cache profiler. It performs detailed simulation of the I1, D1 and L2 caches of the CPU and can pinpoint the sources of the cache misses inside the code. KCachegrind is able to show the number of cache misses, memory references and instructions accruing to each line of source code for every function and for the whole program in a graphical and easier to understand form.

3. **VTune Performance Analyser:** This is a tool by Intel to identify the performance bottlenecks and to improve execution speed of the program. It uses sampling to gain an accurate representation of the actual performance, with negligible overhead. There is no need to instrument the code i.e. no need to make any changes in the code to profile with this tool. Additionally it provides a graphical interface on Windows platforms for analysis and can collect performance data on a remote Linux system. It drills down to the source to identify problematic lines of code and can gather CPU snapshots to identify problems such as cache misses. It provides optimization features as time and event based sampling, hotspot analysis, call graph profiling etc.

4. **HPCMon:** It is a free utility from Intel, a command line tool. It provides a simple interface to the hardware counters for capturing floating point instructions, cache misses etc. No instrumentation of code is required.

Chapter 5

Results (Single Processor)

This chapter presents the results for the single processor implementation. As the main focus was on performance of the 3D implementation, all the results discussed and compared in this chapter are for the 3D case and after implementation of all optimization strategies for the COLBA. PAPI and kcachegrind were used in the initial stages of the implementation, mainly for the 2D case. The following sections discuss profiling results from HPCMon and VTune, and benchmarking on different RRZE systems for 3D case.

Initially, the testing was done for 2D COLBA and iterative algorithm which had shown positive results for COLBA. The result for the cache misses of 2D COLBA and iterative version profiled by PAPI is discussed in Chapter 4, Fig. 4.1. When the same 2D algorithms were profiled on VTune and HPCMon, COLBA showed 99% cache hits as compared to 85% cache hits of iterative algorithm. But for 3D case, COLBA showed 92-93% cache hits as compared to 85-87% cache hits of iterative algorithm. The following measurements were done for mainly analysing 5-6% loss of cache hits. Besides this COLBA performance is equivalent to the optimized LBM kernel at RRZE.

5.1 Case Study

In this section, the analysis of three different LBM kernels. viz. COLBA, iterative 2-way blocked LBM and the fastest LBM kernel at RRZE are discussed. The readings in Tab. 5.1 shows various measurements done with VTune for the 3 different LBM kernels. The terminology used in the further discussion and in the theoretical analysis is described below.

LBMKernel.gw: The optimized LBM kernel at the RRZE with optimal data structure and without blocking (omitting bounce back).

iterative-2way-blocked: The iterative version with (0:18,i,j,k) data layout with 2D blocking (25^2).

5.1.1 Test Case

Domain size : 100^3.

Time steps : 10.

Total Lattice site updates : 100^3 x 10 = 10^7

The theoretical analysis is as follows:

1. To load and store the 19 distribution functions per cell update over the complete domain and total number of time steps: 1.9 x 10^8 loads and 1.9 x 10^8 stores are needed.
2. In LBMKernel.gw, the inner computation has been split into several (3) loops and temporary arrays are used to hold the intermediate results. Thus, in particular, the number of loads should be much higher than in the above estimate.
3. Estimated bus accesses for LBMKernel.gw (no temporal blocking) (1 bus access=8 Words):
   Write Accesses: (1.9/8) x 10^8 = 2.38 x 10^7
   Read Accesses (loads + read for ownership (RFO) request):
   (3.8/8) x 10^8 = 4.76 x 10^7
4. Estimated DTLB misses for LBMKernel.gw (no temporal blocking) (Page size=4 kB):
   - **Total Amount of Memory:** Considering the streaming structure of LBMKernel.gw the total amount of data transfered between CPU and memory sums up to 38 Words x 8 Byte/Word x 10^7 $\approx$ 3.2 GB.
   - Again considering the streaming structure of LBMKernel.gw, the estimate for the DTLB misses is as follows:
     3.2 GB/4 KB $\approx$ 0.8 x 10^6 DTLB misses.

	LBMKernel.gw	iterative-2way-blocked	COLBA
Performance MLUps:	4.9	3.6	4.9
Measurements with vtune			
Bus write	2.34 x 10^7	2.3 x 10^7	1.6 x 10^7
Bus read	5.14 x 10^7	6.7 x 10^7	2.9 x 10^7
DTLB misses	4.0 x 10^6	7.8 x 10^6	2.4 x 10^7
STOREs	0.267 x 10^9	0.37 x 10^9	0.4 x 10^9
LOADs	1.15 x 10^9	0.46 x 10^9	0.7 x 10^9
L1 ld miss	0.11 x 10^9	—	0.11 x 10^9
L2 ld miss	0.02 x 10^9	—	0.012 x 10^9

Table 5.1: Comparison among 3 LBM kernels. Performance measured on Nocona

5.1.2 Observation

DTLB misses are very high for the COLBA. For LBMKernel.gw they are of a factor of 6 lower. So the question arises, about the penalty of a DTLB misses with 2 consequences, if the page table can be found in L2 or it must be loaded from main memory.

The bus reads and writes are in good agreement with the theoretical estimate of LBMKernel.gw. COLBA reduces the number of bus accesses by 30-40%. So it is interesting to understand, how COLBA access the data blocks and what level of temporal blocking is implemented.

As expected the number of loads are much higher than the theoretical estimated loads for LBMKernel.gw. But the store instructions match the estimate very well. Even though COLBA reduces the number of loads by $\approx$ 30%, the number of L1 ld misses is the same as with LBMKernel.gw. The number of L1 misses seem to be rather high. If we assume that they block execution and assume a penalty of 20-30 cycles (L2 latency is 18-20 cycles on Nocona) approx. 20-30 L1 misses are as painful as 1 L2 miss.

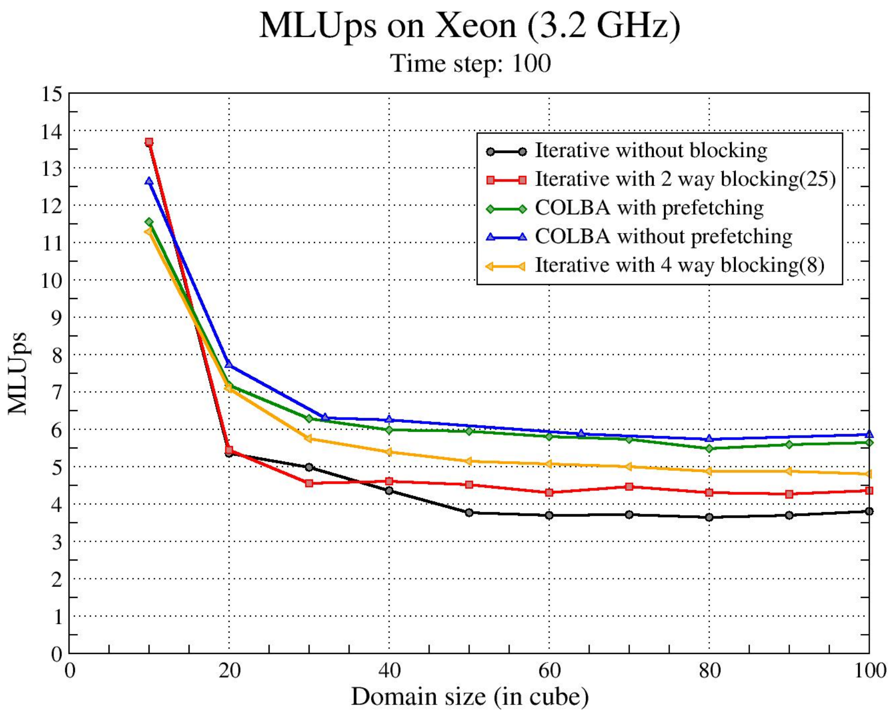
5.2 Profiling with HPCMon

Fig. 5.2 and 5.1 show the different parameters measured with HPCMon tool on Sfront03 (App. A) for four different LBM algorithms. viz. COLBA with and without prefetching and iterative algorithm with 2-way blocking and without blocking.

In the lower chart of Fig. 5.1, at a domain size of 10^3 the complete problem fits inside the cache so all the algorithms show less and almost equal cache misses which are only because of the compulsory cache misses. As the domain size increases, cache misses for all algorithms increase. In case of the Iterative versions, cache misses are high because of cache thrashing. Besides that, 2-way blocked algorithm shows less cache misses as compared to pure iterative algorithm because 2-way blocking increases spacial locality. For both the COLBA, the cache misses are substantially less. The ability of COLBA to explore temporal and spatial locality, keeps cache misses as low as possible. Additionally, COLBA with prefetch shows lesser cache misses, as it tries to prefetch more blocks inside cache.

In the upper chart of Fig. 5.1, at a domain size of 10^3, the performance is high because of very less cache misses. The performance drops after domain size 10^3 until the problem size of 30^3, where it reaches a constant level up to the maximum problem size of 100^3. It is obvious that the drop in performance of iterative version is much more compared to the COLBA. Again, it can be easily mapped to the 'L2 cache misses' graph and justifiable that after domain size of 30^3, there is bigger gap in cache misses of COLBA and iterative algorithm.

In the upper chart of Fig. 5.2, both the COLBA show considerably less bus access compared to the iterative versions. This has been accepted as COLBA does temporal and spacial blocking, most of the data can be found in the cache which reduces the traffic between the main memory and the caches. This gives COLBA an advantage to overcome latency within



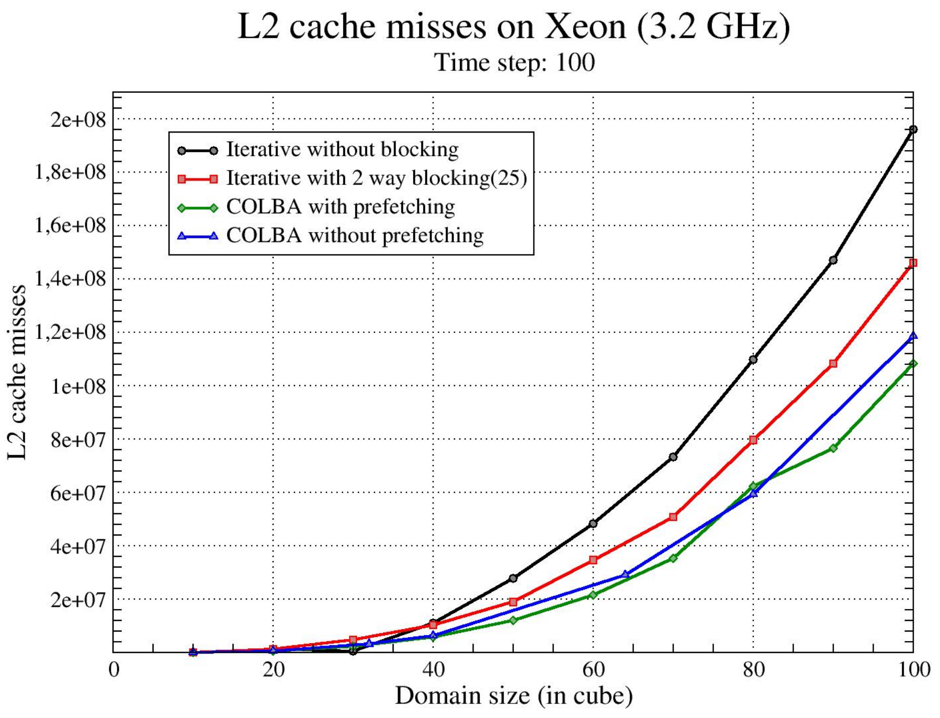


Figure 5.1: Profiling of the COLBA and the iterative version on Sfront03 with HPCMon.

memory transfer and adds up to its performance.

In the lower chart of Fig. 5.2, DTLB Vs domain size, the COLBA shows more DTLB misses compared to any other algorithm. DTLB (Data Translation Lookaside Buffer) is a hardware unit that assists the translation of virtual addresses into physical memory addresses. DTLB miss occurs when the page is not available in the main memory. DTLB misses are much more expensive than L2 cache misses because the access time to primary memory is much longer than even L2 cache.

Thus, from the graphs in Fig. 5.1, 5.2 and the discussed case study , it can be concluded that the DTLB misses is one of the cause for not getting 99% cache hits for the COLBA.

5.3 Result on RRZE Systems

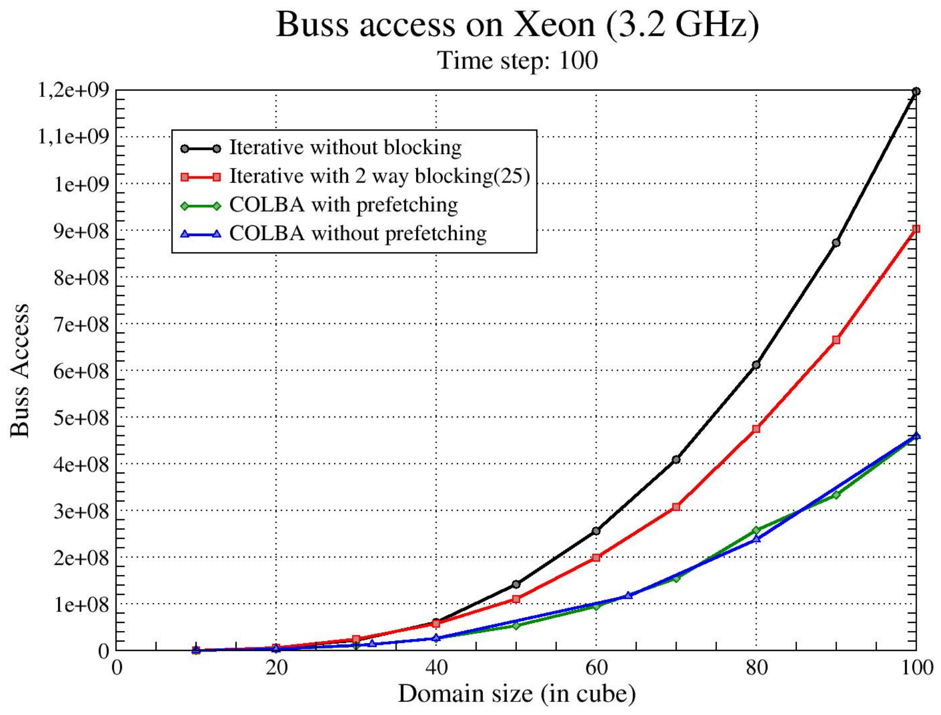
This section presents results on different RRZE test systems which are listed in App. A. The COLBA was benchmarked on different systems available at RRZE, in order to check the performance on different architectures viz. AMD Opteron, Intel's Core 2 and Intel's NetBurst architectures. The results are shown in the upper graph of Fig. 5.3. The AMD architecture (Hammer) shows less performance compared to the two Intel architectures. The performance on Intel's newly emerging Core 2 architecture (woodcrest) is higher than the old NetBurst chips (dempsey1, nocona1, pollux). The lower graph of Fig. 5.3 shows comparison of COLBA and 2-way blocked iterative version on different architectures. The COLBA shows better performance on all architectures.

When COLBA was tested on Itanium2 processor it showed very poor performance. Tab. 5.2 shows the performance for COLBA on MC-2.

Domain size (in cube)	10	50	100	200
MC-2	6.86	2.12	1.3	–
Altix	–	–	1.6	1.53

Table 5.2: Performance in MLUps for 100 time steps.

Itanium2 always needs long loop length for efficient software pipe lining. The optimum loop length should be greater than 100. In case of the COLBA the loop length is very small that is 3-4 because COLBA works on small patches (3^3 or 4^3). Even though the COLBA utilizes cache effectively because of Itanium2 architecture performance drops down heavily.



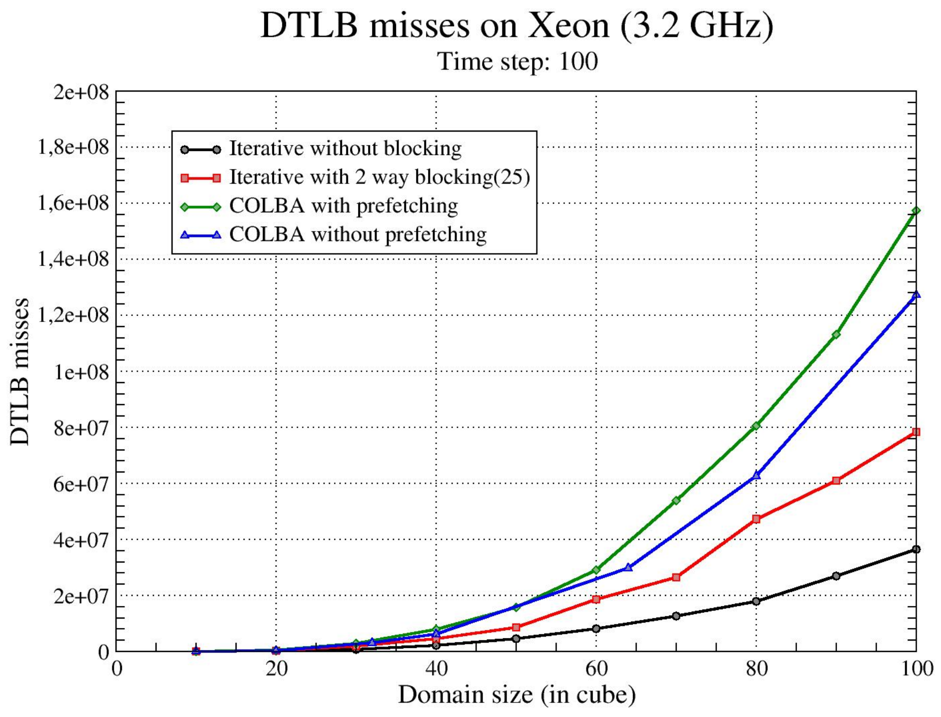
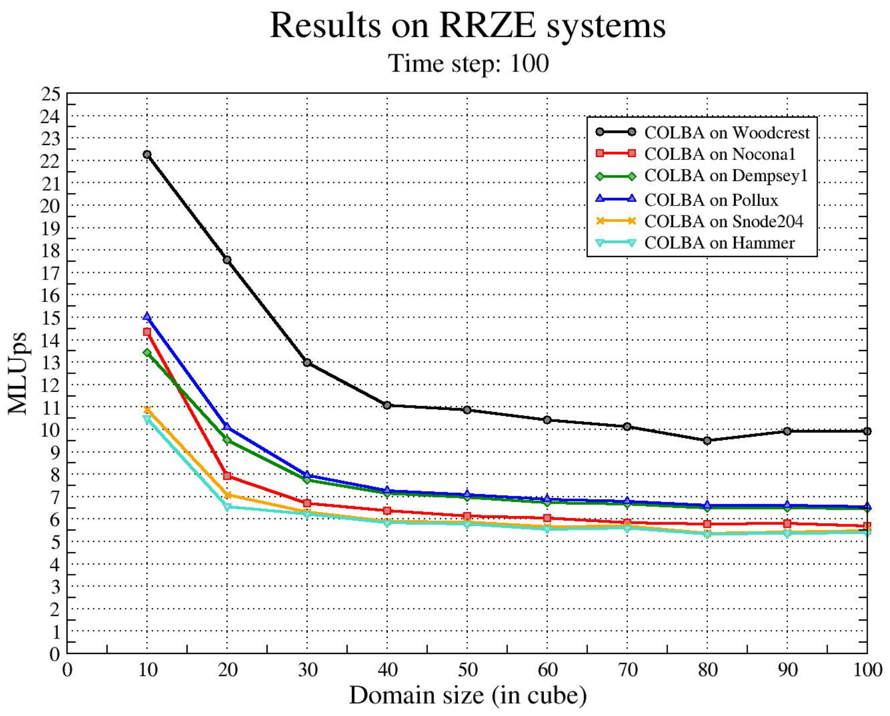


Figure 5.2: Profiling of the COLBA and the iterative version on Sfront03 with HPCMon.



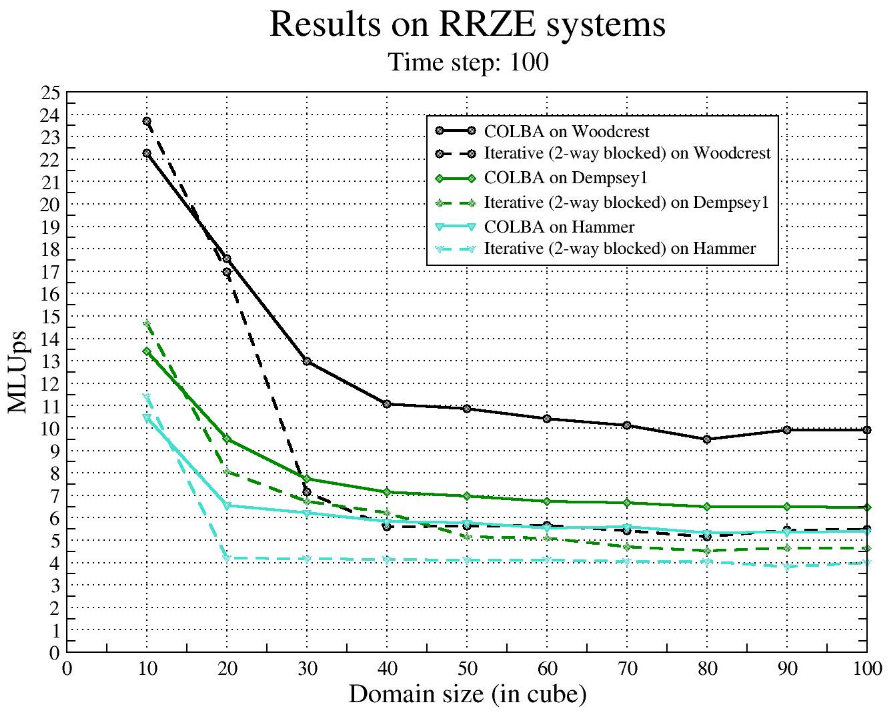


Figure 5.3: Performance measurement on RRZE test systems.

Part III

Multiple Processor Implementation

Chapter 6

Parallelization for Shared Memory Architectures

Next to the single processor implementation, a shared memory parallel implementation was developed. The focus of the implementation was to test the adaptability of the COLBA for parallel architectures using OpenMP. The long term idea behind the parallelization effort has been the use of the COLBA for each node of a cluster, while using a standard MPI parallelization between the nodes. This hybrid parallelization should then be possible for each LBM simulation on a cluster system.

Due to the recursive nature of the COLBA, the parallelization using OpenMP was challenging. The initial idea was to make the COLBA suitable for multi-threading, which means that some parallel directives are added inside COLBA itself. It proved to be difficult to fit OpenMP directives inside the logic of the COLBA. Additionally, the successive recursive function calls makes the OpenMP directive as 'orphan'. Inside the orphan directive OpenMP may or may not spawn threads. However, the decision over the spawning of new threads is not under the control of the programmer. Obviously, OpenMP is more suited for the parallelization of standard *for* loop structures without recursion; in the case of the COLBA, no new threads were spawned.

6.1 The Parallelization Approach

The main idea of parallelization approach is based on a logical domain decomposition of the space-time domain as shown in Fig. 6.1. Thereby each independent domain is solved in parallel with COLBA. The 4D space-time domain is always divided into small domains in the shape of parallelograms and triangles. Using these shapes the dependencies in the stencil calculations of LBM kernel can be resolved. Each parallelogram or triangle will have a slope -1 for its slant edge. The division of the space-time domain is purely virtual, which means that all regions lie in the same shared memory and all processors can access any virtual domain. Therefore the communication overhead among the processors is eliminated on a shared memory system. The illustration on the right in Fig. 6.1 has inclined lines on the small blocks. Thus, the blocks along each line can be solved in parallel. The complicated task for this approach is to ensure that bottom and left side blocks should be finished before starting any new individual block in order to resolve the data dependancy in the computation.

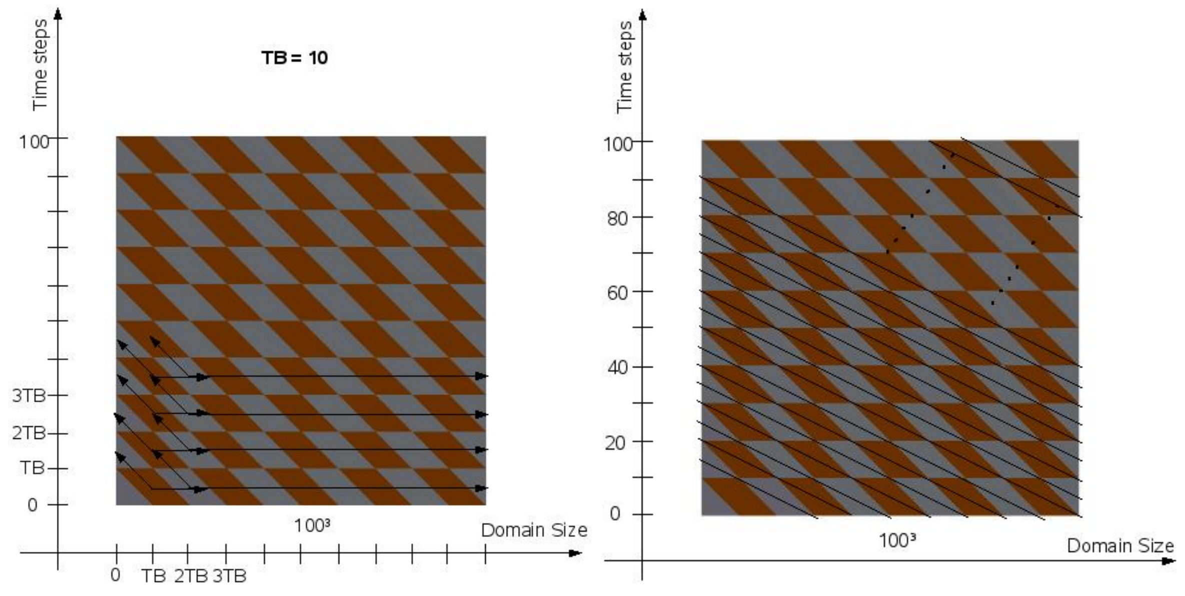


Figure 6.1: Thread spawning in virtually decomposed parallel regions.

6.2 Thread Spawning

Before discussing the thread spawning, the **Time Blocking Factor (TB)** has to be introduced. It decides how many time steps are solved within each domain. It must divide the domain size and the total number of time steps completely (remainder should be zero). In Fig. 6.1, the TB is set to 10. In order to maintain the slope equal to -1, all triangular and parallelogram domains should be of size 10 and can be solved for 10 time steps with one COLBA update.

The threads are always spawned in the domains within TB to 2TB and 2TB to 3TB. Once a thread is spawned, it solves all domains in the same row as illustrated in left diagram of Fig. 6.1. When thread reaches the triangular domain, then it is terminated after solving the triangular domain. As can be seen in the left diagram of Fig. 6.1, every thread is going to come across a triangular region and therefore it is guaranteed to be terminated.

The traversal of this parallel algorithm can be visualized as a tree as shown in the left illustration of Fig. 6.2. Each node represents one domain, which is either a triangle or a parallelogram. On the left hand side of the tree, there are long branches, which represent rows and are terminated when the calculation of the row is finished. The numbers inside the tree nodes represent the corresponding triangular or parallelogram domain in the rectangular space-time region.

Alg. B.6 and Fig. 6.2 show the implementation of this idea. The *parallelWalk()* function is called for each subdomain by the corresponding thread, which internally calls the COLBA. Again, this function is recursive in nature, but by using Intels taskq model the orphan directives can be overcome.

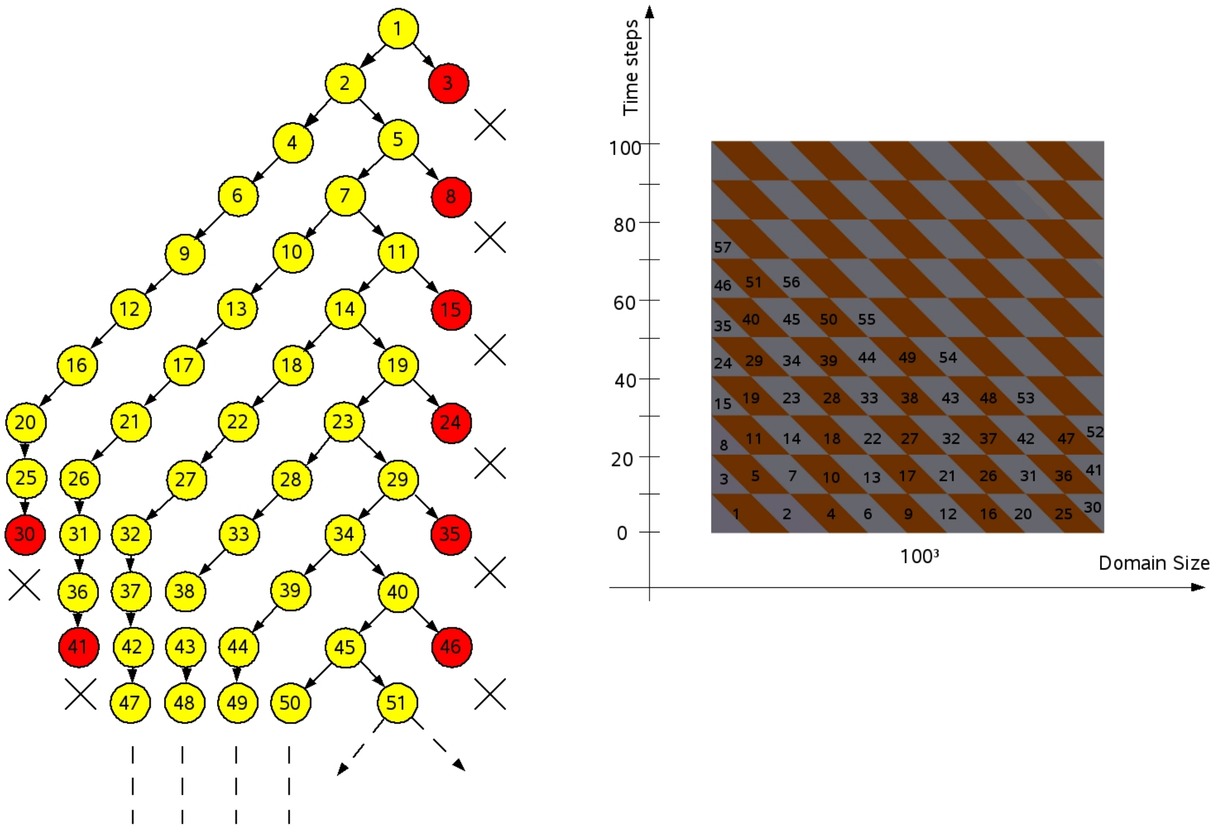


Figure 6.2: The mapping of virtually decomposed domains to a tree.

6.3 Parallelization with Intel's Taskq Model

The Intels task queue model is a shared memory parallelization extension of the Intel compiler. This model is especially useful for the parallelization of *while* loops and recursive function calls. It queues the function calls and then solves the queued items in parallel. For this, the special pragma *#pragma intel omp taskq* and *#pragma intel omp task* are used. The placement of both pragmas is similar to the *section* directive in the OpenMP set. The threads are spawn inside each *#pragma intel omp task* which are placed inside *#pragma intel omp taskq*.

6.4 Locking

As it can be seen in Fig. 6.2, all regions are dependent on their left and bottom side neighbors. Unless the left and bottom neighbors are solved, the dependent domain can not be solved. The thread for this domain has to wait for the other threads to complete their tasks. Therefore, it is necessary to implement locking inside the algorithm.

A 1D array which is called synchronization table and which has a length equal to the number of virtual domains was implemented in such a way that the synchronization table has one lock entry per domain. From an implementation point of view, a 1D array of *omp_lock_t* variables was created and mapped to a 2D access. Initially, all locks are set to 'true', i.e all regions are locked. A specific domain can only be unlocked by the corresponding thread. Before solving its own domain, each thread checks whether the left and bottom domains are unlocked. If both neighbors have already been unlocked, then it will first solve and then unlock its own domain, otherwise it will wait, until both neighbors have been solved.

Listing 6.1: COLBA : Multi(N)-Processor implementation.

```c
void solvePara(int t0, int t1, int x0, int dx0, int x1, int dx1, int y0, int dy0, int y1, int dy1, int z0, int dz0, int z1, int dz1)
{
  int loct0 = t0;
  int loct1 = DT; //DT is time blocking factor.
  int locy0 = y0, locy1 = 2*DT;

  walk3D(loct0, loct1, x0, dx0, x1, dx1, locy0, dy0, locy1, -1, z0, dz0, z1, dz1);

  omp_unset_lock(&syncTable[0]);
  omp_unset_lock(&syncTable[1]);

  #pragma omp parallel sections
  {
    #pragma omp section
    {
      walk3D(loct0+DT, loct1+DT, x0, dx0, x1, dx1, locy0, dy0, DT, -1, z0, dz0, z1, dz1);
      omp_unset_lock(&syncTable[(((loct0+DT)/DT) * stCol) + ((locy0)/DT) ]);
    }
    #pragma omp section
    {
      walk3D(loct0, loct1, x0, dx0, x1, dx1, locy1, -1, locy1+DT, -1, z0, dz0, z1, dz1);
      omp_unset_lock(&syncTable[((loct0/DT) * stCol) + ((locy1)/DT) ]);
    }
  }

  #pragma omp parallel
    #pragma intel omp taskq
      #pragma intel omp task
        parallelWalk(loct0, loct1, x0, dx0, x1, dx1, locy1, -1, locy1+DT, -1, z0, dz0, z1, dz1);

}
```

Algorithm 6.1 ParallelWalk function

```
void parallelWalk(int t0, int t1, int x0, int dx0, int x1, int dx1, int y0, int dy0, int y1, int dy1, int z0, int
dz0, int z1, int dz1)
if dy0 == 0 || dy1 == 0 then
   return
else
   if t1 != totalTs then
      if y0 == DT && y1 == 2*DT then
         #pragma intel omp taskq
         #pragma intel omp task
         if t0 ≥ DT (Not bottom most (1st row) parallelogram) then
            Check the dependant domains are unlocked.(left and bottom parallelograms)
            Solve the Domain.
         else
            Solve the domain.
         end if
         Unlock the parallelogram.
         Call parallelWalk()
         #pragma intel omp task
         Solve the triangular domain.
         Unlock the triangle.
         Call parallelWalk()
         return
      else if y0 == 2*DT && y1 == 3*DT then
         # pragma intel omp taskq
         # pragma intel omp task
         if t0 ≥ DT (Not bottom most (1st row) parallelogram) then
            Check the dependant domains are unlocked.(left and bottom parallelograms)
            Solve the Domain.
         else
            Solve the domain.
         end if
         Unlock the parallelogram.
         Call parallelWalk()
         #pragma intel omp task
         Solve the parallelogram domain.
         Unlock the parallelogram.
         Call parallelWalk()
         return
      else if y1 == COLSIZE then
         Solve triangles in last columns.
         Unlock the triangle.
         return
      else
         // Solves the central domain.
         SEE THE CONTINUATION.
         return
      end if
   else
      SEE THE CONTINUATION.
   end if
   return
end if
return
```

Algorithm 6.2 ParallelWalk function continuation...

```
if dy0 == 0 || dy1 == 0 then
    ...
else
    if t1 != totalTs then
        if y0 == DT && y1 == 2*DT then
            ...
        else if y0 == 2*DT && y1 == 3*DT then
            ...
        else if y1 == COLSIZE then
            ...
        else
            if t0 $\geq$ DT then
                Check the dependant domains are unlocked.(left and bottom parallelograms)
                Solve the Domain.
            else
                Solve the domain.
                Unlock the parallelogram.
                Call parallelWalk()
            end if
            return
        end if
    else
        if y1 == COLSIZE then
            Solve right hand side top most triangular domain.
            Unlock triangle. // But it won't affect because it is the last region.
            return
        else
            // check the bottom region; it should be unlock.
            Check the dependant domains are unlocked.
            Solve the top most Domain (row).
            Unlock the parallelogram // it won't affect because it is top most row.
            Call parallelWalk()
            return
        end if
    end if
    return
end if
return
```

Chapter 7

Results (Multi-Processor)

7.1 Benchmarking of Parallel COLBA

This section presents benchmarking results for parallelization on systems listed in Appendix A. The performances on the different multi-processor systems are shown from Fig. 7.1 to Fig. 7.5. The performances are measured by,

- Varying number of time steps and keeping domain size and number of threads constant.
- Varying number of threads and keeping domain size and time steps constant.
- With three time blocking factors viz. 10, 20 and 25 for all benchmarking.

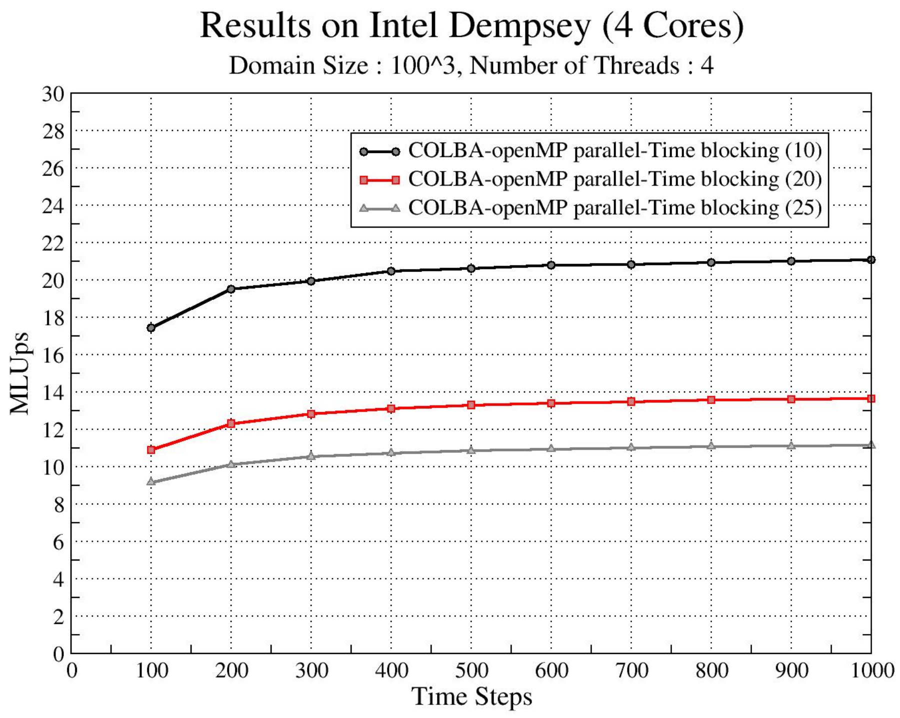
7.2 Observations

7.2.1 Effect of Time Blocking Factor

The time blocking factor (TB) is introduced in section 6.2. It decides the effective parallel regions. **Effective parallel region** is defined as the maximum parallelization achieved during multi-processor execution. The approximate formula to calculate effective parallel regions (epr) is

$$epr = \frac{\frac{\text{domain size}}{\text{time blocking}}}{2}$$

For domain size of 100^3 and 100 time steps, with TB equal to 10, 5 or 6 effective parallel regions are created while with time blocking of 25, only 2 or 3 effective parallel regions are created. Fig. 7.6 illustrates how the number of parallel regions are changed by changing the time blocking factor. The two inclined lines show the effective parallel region. It affects the performance significantly. It can be seen from the graphs in Fig. 7.1 to Fig. 7.5, time blocking factor 10 gives best performance compared to 20 and 25. In case of Hammer(8 cores), time blocking factor 5 is also used which creates 10 parallel region and gives better performance over TB=10. For other testing systems with 2 or 4 cores, time blocking factor 5 did not show gain in performance because it creates more parallel regions than total number of processors. Thus, in this case processor capacity limits the performance.



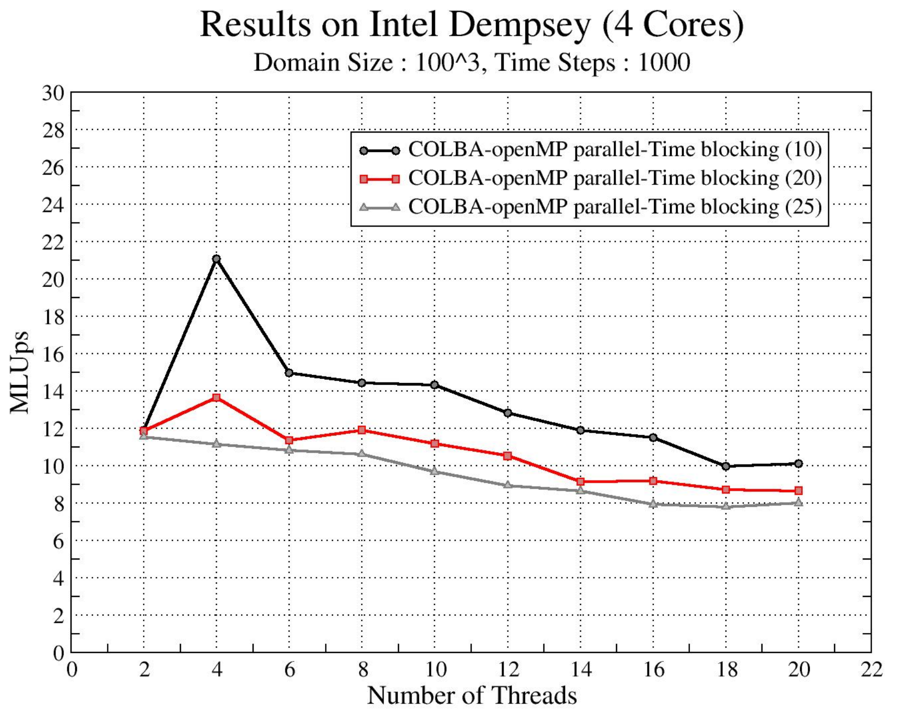
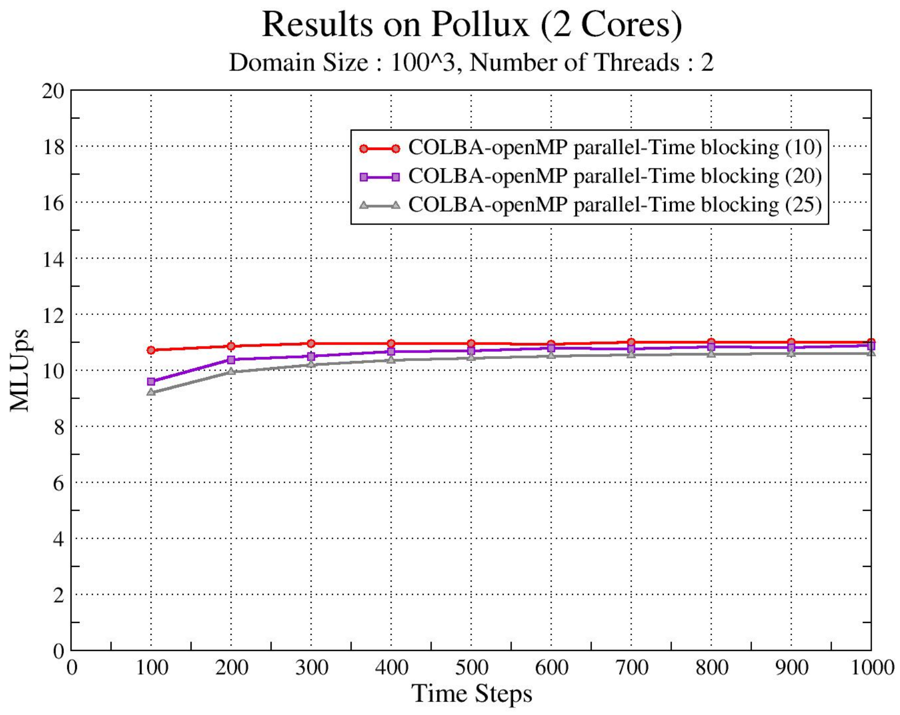


Figure 7.1: Performance measurement on Dempsey.



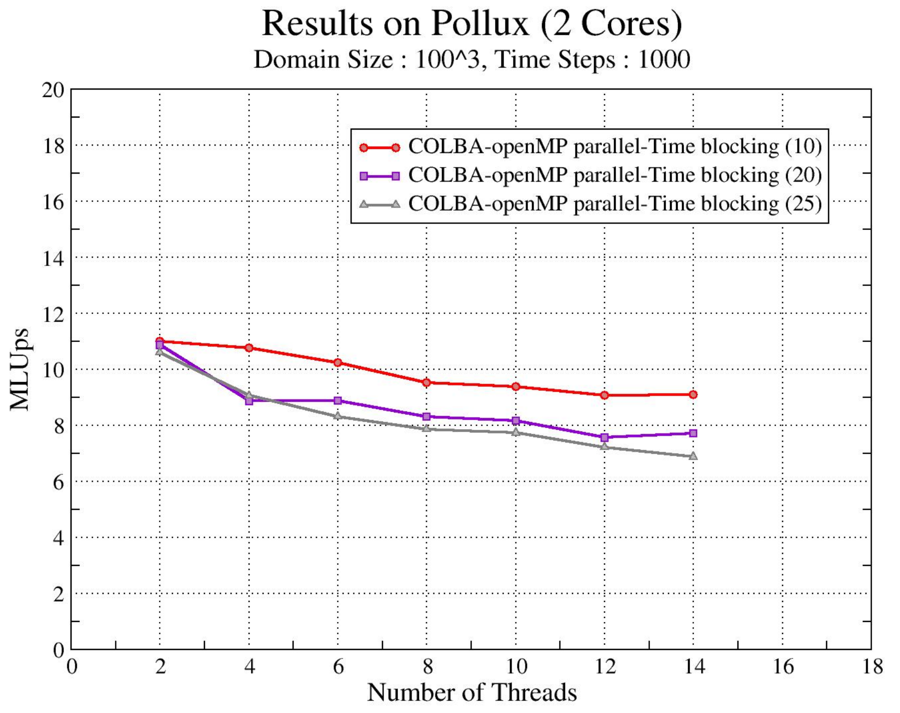
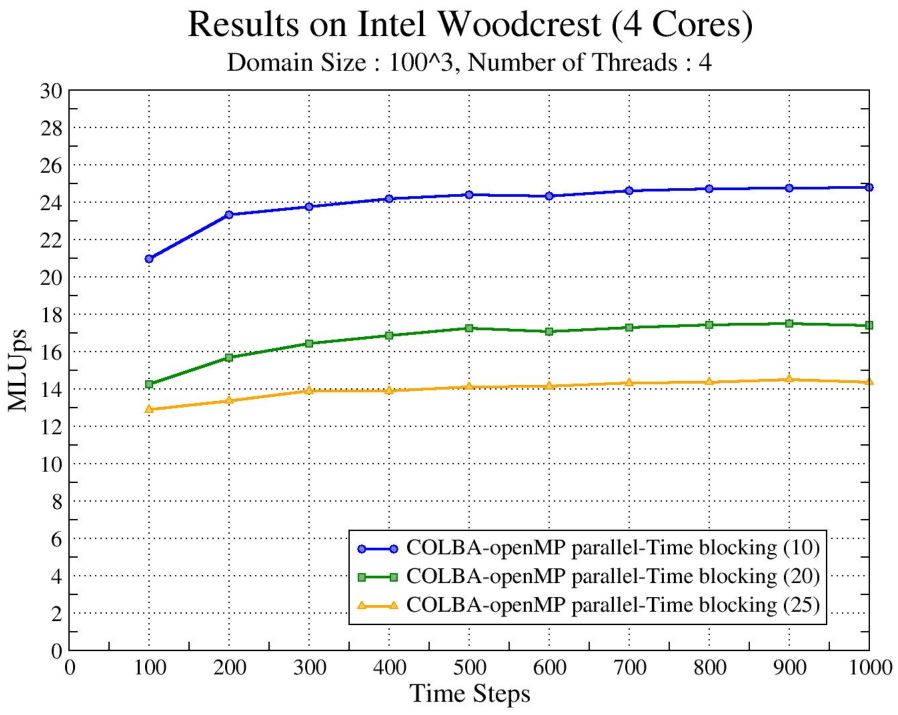


Figure 7.2: Performance measurement on Pollux.



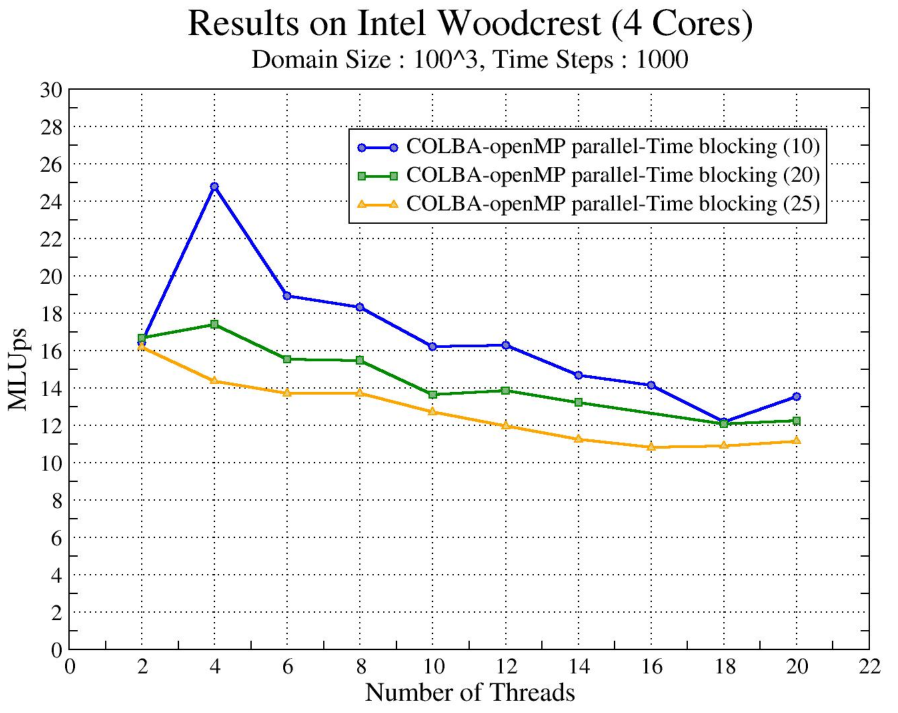
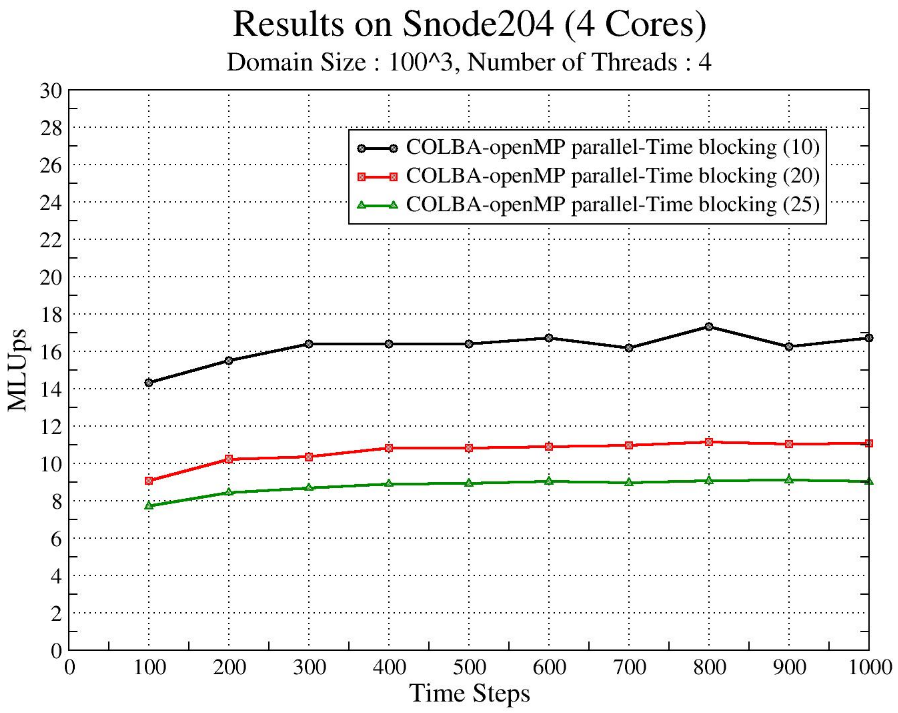


Figure 7.3: Performance measurement on Woodcrest.



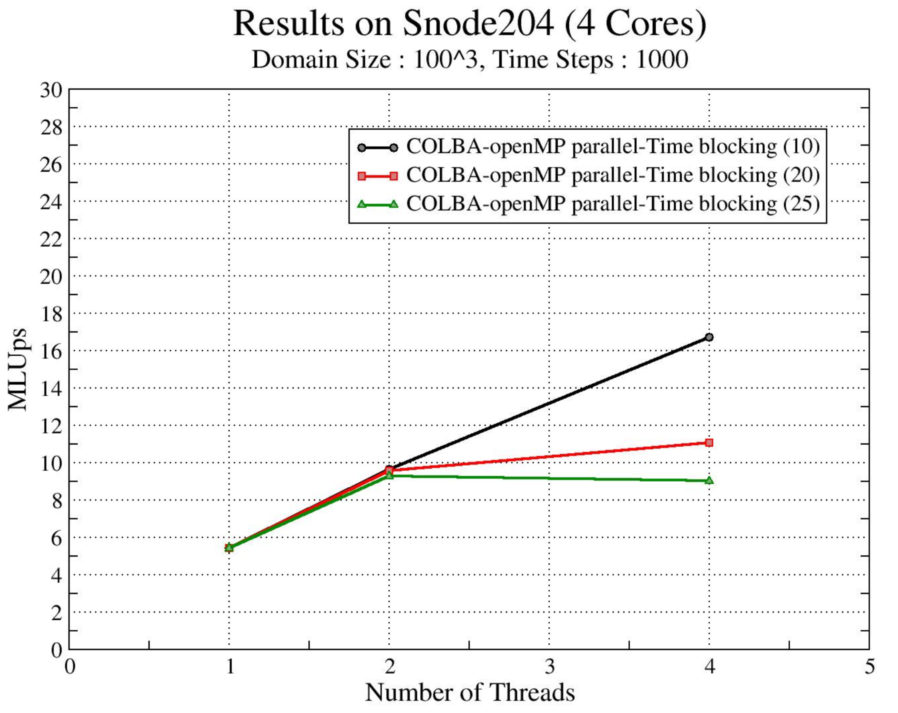
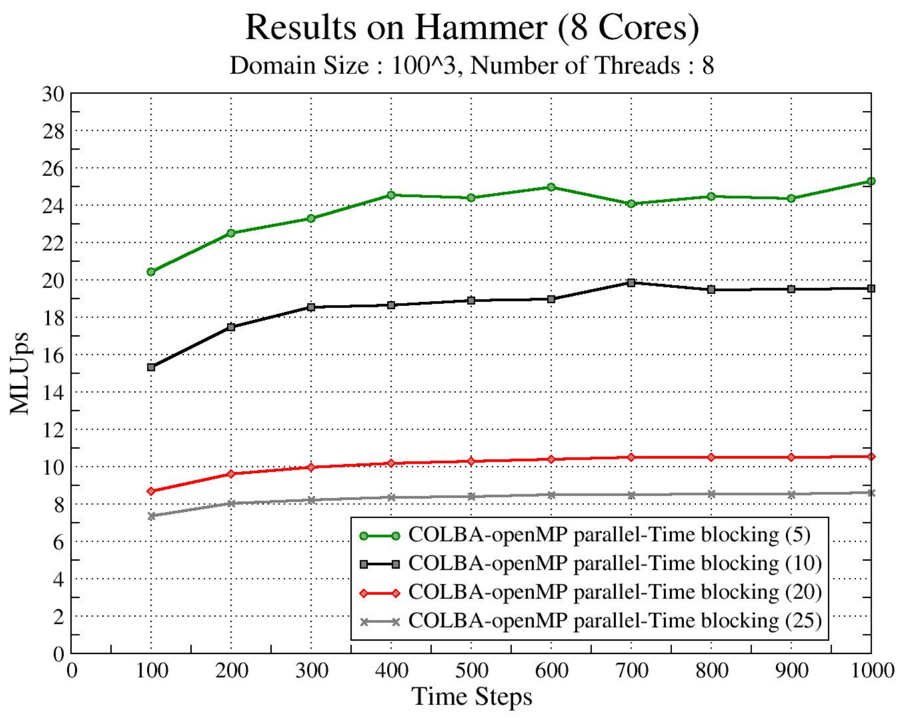


Figure 7.4: Performance measurement on Snode204.



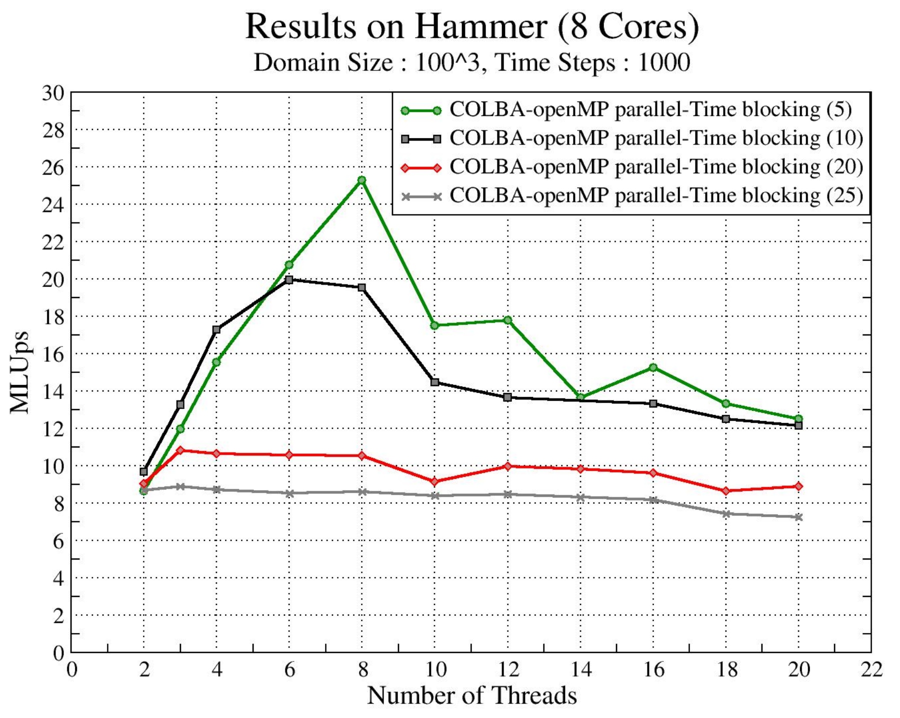


Figure 7.5: Performance measurement on Hammer.

Remark:

1. It is observed that the best performance on any system is achieved when number of threads equal to number of processors equal to number of effective parallel regions.
2. Increase in number of threads not necessarily increases the performance. This can be observed from Fig. 7.1 to Fig. 7.5, until the number of threads are less than number of processors the performance increases, but if number of threads crossed the total number of processors, then the performance drops. Excessive threads might be creating load imbalance among the threads.

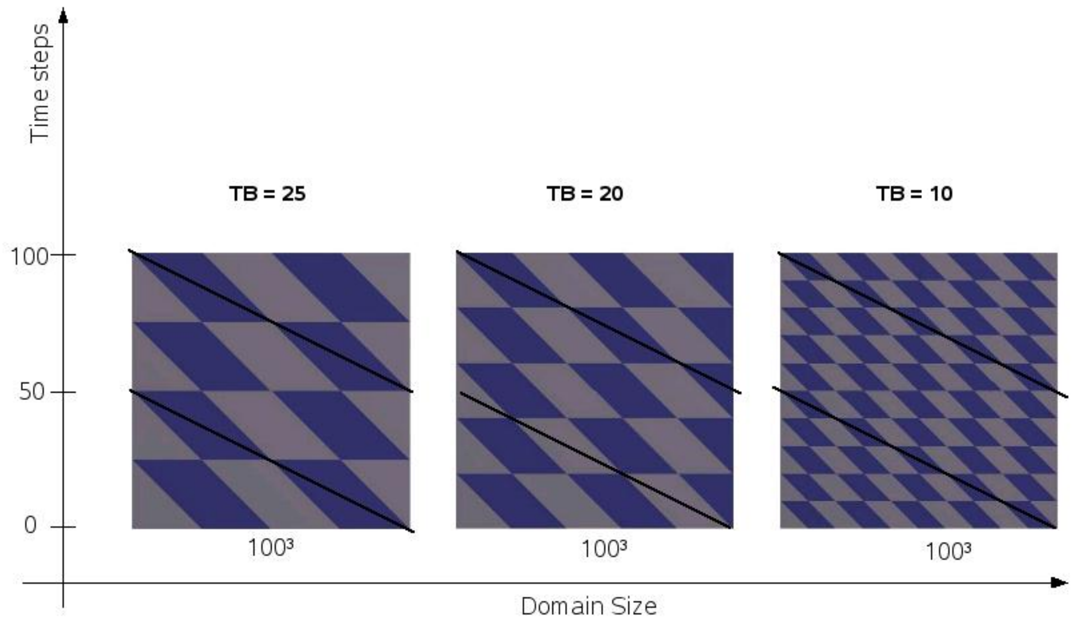


Figure 7.6: Effect of time blocking factor.

7.2.2 More Parallelization with more Time Steps

It can be seen from the graphs from Fig. 7.1 to Fig. 7.5, that as time steps increase the performance (MLUps) of the parallel COLBA increases gradually. This can be explained with the help of Fig. 7.7. It shows three pictures with an increasing number of time steps. The effective parallel region is marked with two inclined lines in each picture. Thus, domain size with 200 time steps shows more parallel region compared to 50 and 100 time steps.

7.2.3 Comparison of 32-bit and 64-bit Compilation

The new 64-bit Intel 9.1 compiler takes best advantage of multi-core Intel processors. It has some advanced optimization features such as OpenMP specific support for multi-threaded application, automatic vectorizer and some high level aggressive optimization with loops and prefetching etc. In Fig. 7.8 The performance of 32-bit Intel 9.0 compiler is compared to the

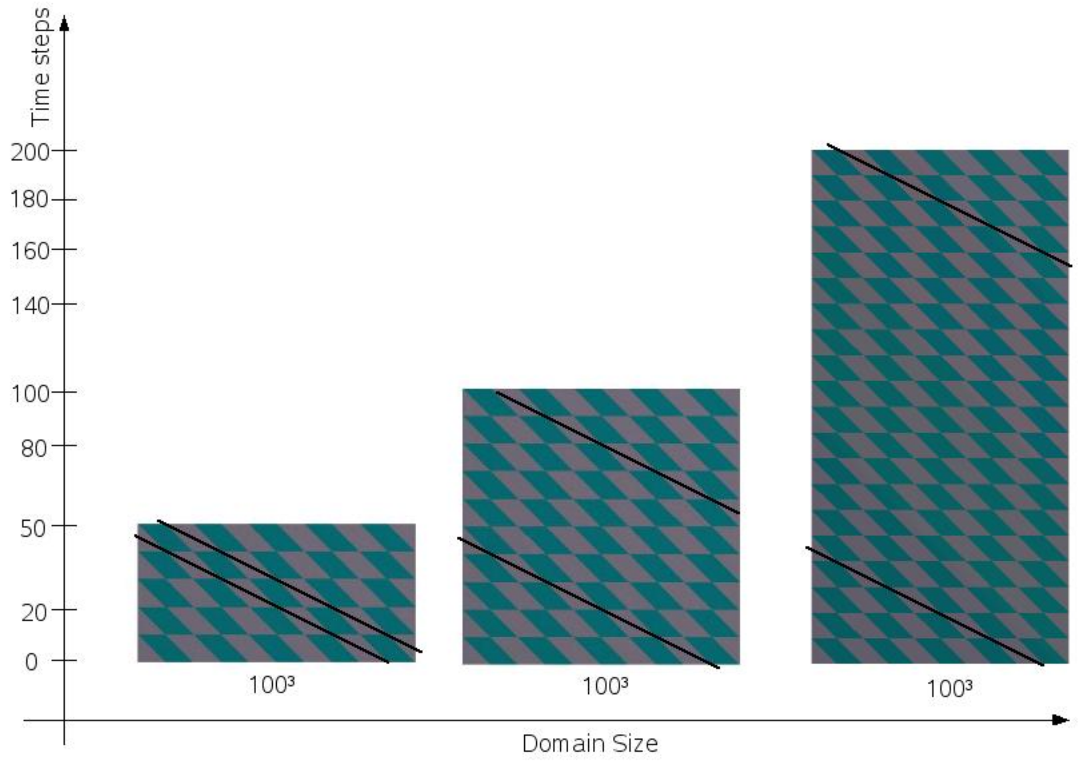


Figure 7.7: The parallel region increases with increase in time steps.

performance of 64-bit Intel compiler. The 64-bit Intel compiler could not increase the performance of the COLBA. The testing has been performed on multi-core Woodcrest processor with 4 threads running in parallel.

7.2.4 Comparison of Parallel COLBA and Iterative Version

As the reference for the measurements, the single processor iterative version was also extended to multi-processor, parallel version by using OpenMP. The performance for both multi-processor implementation viz. COLBA and iterative, is shown in Fig. 7.9. Originally, the iterative single processor algorithm is much slower than single processor COLBA so iterative parallel version also shows less performance than parallel COLBA.

7.2.5 Testing on 2 Cores, in the Same Socket and in the Different Sockets

With the help of the *taskset* command, the number of cores can be defined so that the program can execute on the particularly defined cores. In Fig. 7.10, a performances comparison between woodcrest and snode204 with 2 threads is shown. There it can be seen that both the systems show higher performance for cores in the same socket. For woodcrest system, choice of socket has more impact on performance compared to snode204 system. Also, the performance for cores in same socket and without specifying the cores was same. Thus, for

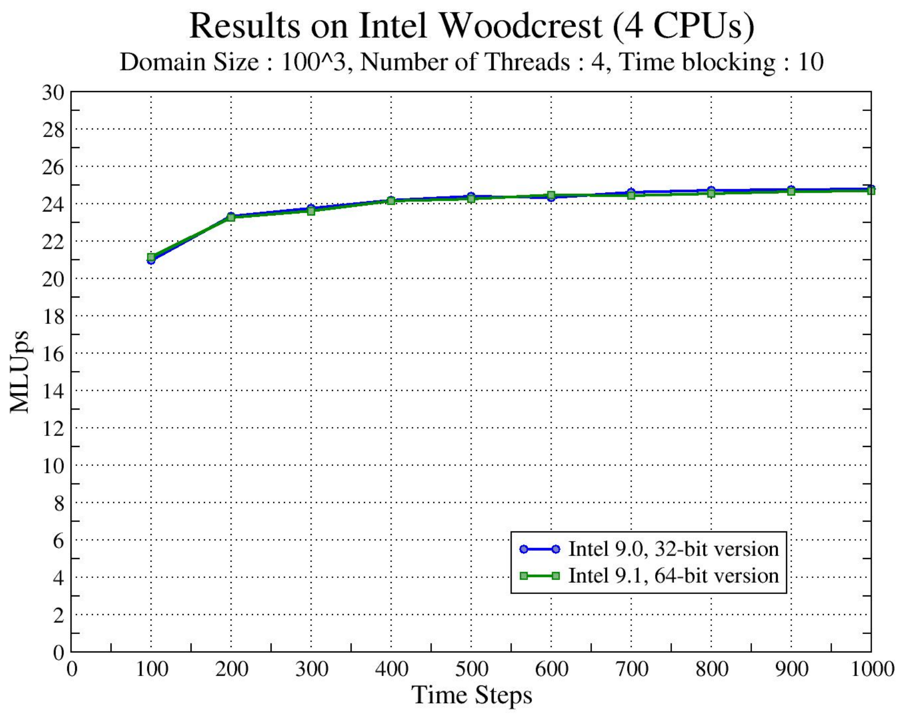


Figure 7.8: Performance measured by compiling the program with two versions of Intel compilers. 32-bit ICC-9.0 compiler and 64-bit ICC-9.1 compiler.

woodcrest system it can be inference that by default the threads run on the cores in the same socket but for snode204 it may not.

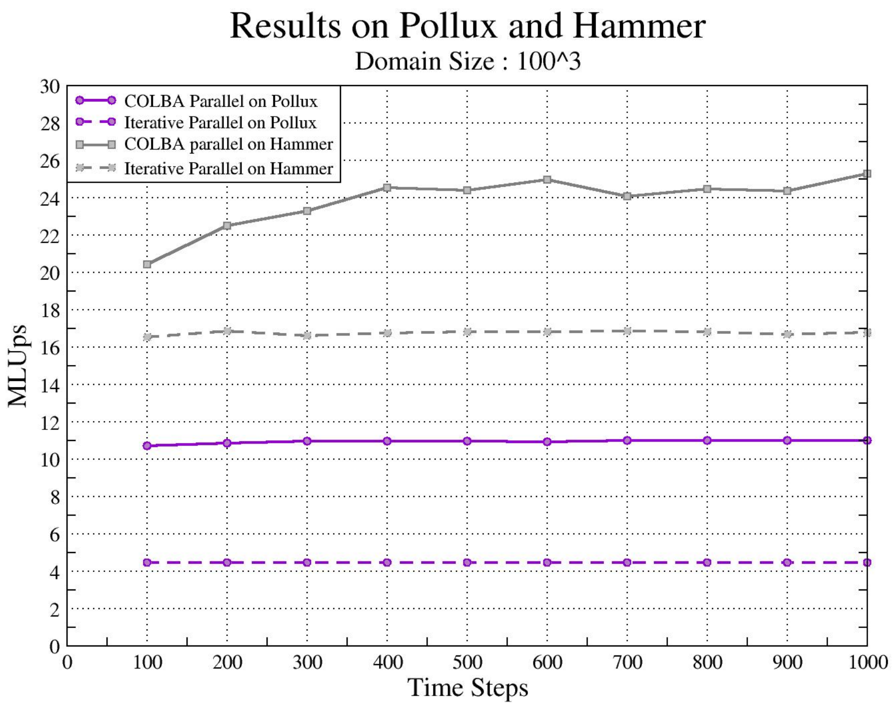
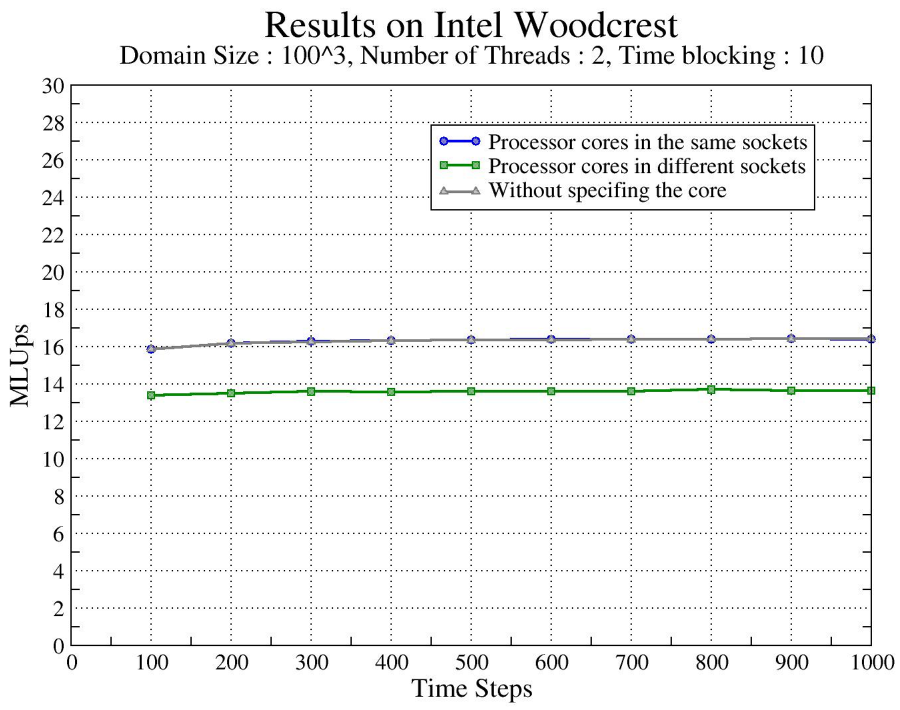


Figure 7.9: Comparison between COLBA and Iterative parallel versions.



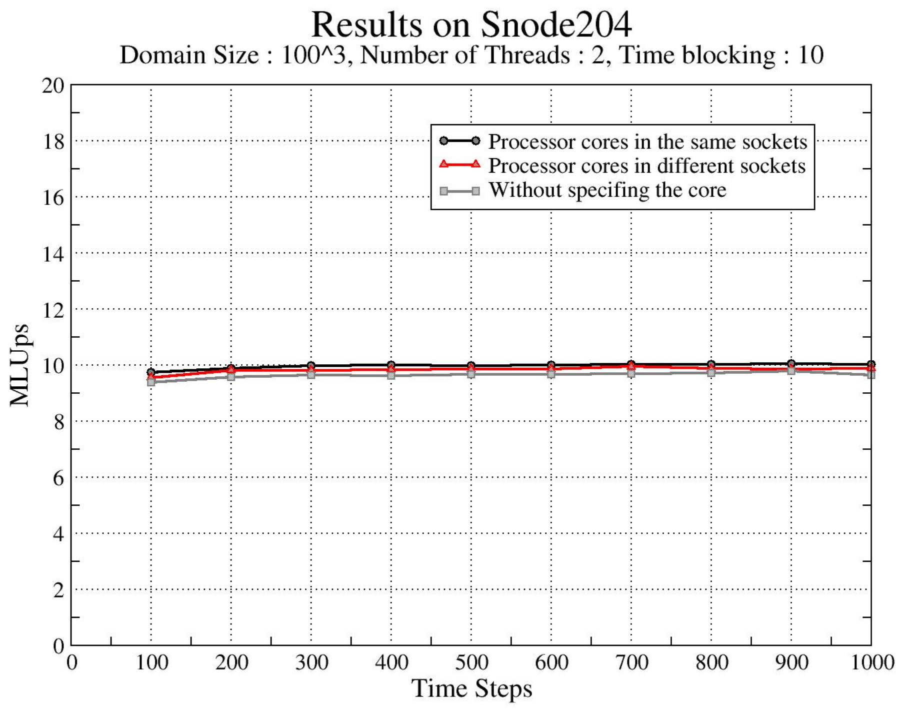


Figure 7.10: Performance measurement by defining and not defining core numbers.

Part IV

Conclusion and Appendix

Chapter 8

Conclusion

In this thesis, the feasibility of a cache oblivious algorithm in the context of the lattice Boltzmann method was shown and it was proved that the efficient cache oblivious lattice Boltzmann algorithm is practicable. The COLBA, in 2D and 3D behaved as expected. Both implementations show substantially less cache misses compared to the iterative implementation of the LBM as well as the performance was equivalent to the fastest LBM kernel at RRZE. Additionally, the COLBA has also been proven to be optimal in the cache based memory hierarchy.

Throughout this thesis, many ideas were tried out during the implementation and the optimization of the COLBA. However, only few approaches, like splitting of the LBM kernel into two functions and the usage of compiler directives proved to be able to yield improvements in performance. In the multi-processor implementation for the shared memory architecture, COLBA was implemented with virtual domain decomposition. The parallelization with OpenMP proved the adaptability of the COLBA with OpenMP. As well as parallelization results show scalability of COLBA with the number of CPUs. The Intel's taskq model which suits best for parallelization of recursive functions was able to overcome the hurdle of 'orphan' directives in OpenMP.

Along with some good features, also some pit falls have been discovered during the benchmarking and the analysing stage of the COLBA. e.g. it showed high DTLB misses compared to iterative algorithm. Ability of the COLBA to explore the cache makes it useless on vector machines. The multi-processor implementation, using *Intel taskq model* makes the scope of parallelization dependent on Intel compiler.

In this thesis, the COLBA has been tested for simple geometry like lid-driven cavity. There is still a wide scope remaining for future implementation and testing of the COLBA to deal with the cache coherent non-uniform memory access (CCNUMA) problem, for handling complex geometries and various boundary conditions. Further parallelization with MPI or OpenMP-MPI, a hybrid approach could also be tested. Thus this thesis breaks new ground for computational development of lattice Boltzmann method.

Appendix A

Bench Marking Systems

A.1 Pollux

- Processor: Intel Xeon EM64T CPU (3.6 GHz) - 2 cores.
- L1 Cache: 16 KB.
- L2 Cache: 2MB.
- Main memory: 4 GB.
- Operating System: Suse linux 9.3.
- L1 associativity: 8-way.
- L2 associativity: 8-way.

A.2 Dempsey

- Processor: Intel Dempsey CPU (3.2 GHz) with Bensley Chipsets - 4 cores.
- L1 Cache: 16 KB.
- L2 Cache: 2 MB.
- Main memory: 4 GB.
- Operating System: Suse linux 10.
- L1 associativity: 8-way.
- L2 associativity: 8-way.

A.3 Hammer

- Processor: Dual core AMD Opteron 875 (2.2 GHz) - 8 cores.
- L1 Cache: 64 KB.
- L2 Cache: 1 MB.
- Main memory: 16 GB.
- Operating System: Suse linux enterprise server (SLES) 9.
- L1 associativity: 2-way.
- L2 associativity: 16-way.

A.4 Woodcrest-Testing System for short duration

- Processor: Intel Woodcrest CPU (3 GHz) - 4 cores.
- L1 Cache: 32 KB.
- L2 Cache: 2 MB/core; 4 MB/2 cores.
- Main memory: 8 GB
- Operating System: Suse linux enterprise server (SLES) 9.
- L1 associativity: 8-way.
- L2 associativity: 16-way.

A.5 Woodcrest

- Processor: Intel Woodcrest CPU (2.66 GHz) - 4 cores.
- L1 Cache: 32 KB.
- L2 Cache: 2 MB/core; 4 MB/2 cores.
- Main memory: 8 GB
- Operating System: Suse linux 10.1.
- L1 associativity: 8-way.
- L2 associativity: 16-way.

A.6 Nocona

- Processor: Intel Xeon EM64T CPU (3.4 GHz) - 2 cores.
- L1 Cache: 16 KB.
- L2 Cache: 1 MB.
- Main memory: 4 GB.
- Operating System: Suse linux 9.1
- L1 associativity: 8-way
- L2 associativity: 8-way

A.7 MC-2

- Processor: Itanium2 (1.4 GHz)
- L1 Cache: NA.
- L2 Cache: 256 KB
- L3 Cache: 1.5 MB.
- Main memory: 10 GB.
- Operating System: Suse linux enterprise server (SLES) 9.
- L2 associativity: 8-way, 16 banks w/16 bytes each
- L3 associativity: 6-way.

A.8 Sfront03

- Processor: Dual Xeon EM64T (3.2 GHz).
- L1 Cache: 16KB.
- L2 Cache: 1 MB
- Main memory: 4 GB
- Operating System: Debian linux 3.1.
- L1 associativity: 8-way
- L2 associativity: 8-way

A.9 VMSNode2

- Processor: Dual Xeon EM64T (3.2 GHz).
- L1 Cache: 16KB.
- L2 Cache: 1 MB
- Main memory: 2 GB
- Operating System: Suse linux enterprise server (SLES) 9.
- L1 associativity: 8-way
- L2 associativity: 8-way

A.10 Snode204

- Processor: Dual core AMD Opteron (2 GHz) - 4 cores.
- L1 Cache: 64 KB.
- L2 Cache: 1 MB.
- Main memory: 4 GB.
- Operating System: Debian linux 3.1.
- L1 associativity: 2-way.
- L2 associativity: 16-way.

A.11 Altix

- Processor: Altix (1.4 GHz)
- L1 Cache: NA.
- L2 Cache: 256 KB
- L3 Cache: 4 MB.
- Main memory: –
- Operating System: Suse linux enterprise server (SLES) 9.
- L2 associativity: 8-way, 16 banks w/16 bytes each
- L3 associativity: 12-way.

A.12 Fauia59

- Processor: AMD Opteron 248 (2.1 GHz).
- L1 Cache: 64 KB
- L2 Cache: 1 MB.
- Main memory: 4 GB
- Operating System: Suse linux enterprise server (SLES) 9.
- L1 associativity: –
- L2 associativity: 4-way

Appendix B

Implemented Code

Algorithm B.1 Lattice Boltzmann Kernel in 3D : Iterative version

```
1: void IterativeLBM(Cell* src, Cell* dst, int x0, int x1, int y0, int y1, int z0, int z1, int t)
2: for timestep = 0 to t by 1 do
3:     // Function to solve Lattice Boltymann Method
4:     LBMKernel3D(src, dst, x0, x1, y0, y1, z0, z1)
5:     // Swapping of two grids
6:     tmpCell = src;
7:     src = dst;
8:     dst = tmpCell;
9: end for
```

Listing B.1: Lattice Boltzmann Kernel in 2D.

```
void LBMKernel2D(Cell* src, Cell* dst, int x0, int x1, int y0,
   int y1, int t0)
{
  double  rho = 0.0, ux = 0.0, uy = 0.0, u_sqr = 0.0, w_0 = 0.0,
      w_1 = 0.0, w_2 = 0.0, rhoinv = 0.0;
  int i = 0,  j = 0;

  for(i=x0;i<x1;++i)
  {
    for(j=y0;j<y1;++j)
    {
      if(0 == i) //  South Boundary
      {
        dst(i,j).C = 0.0;
        dst(i,j).N  = src(i+1,j).S;
        dst(i,j).NE = src(i+1,j+1).SW;
        dst(i,j).NW = src(i+1,j-1).SE;
      }
      else if(0 == j) //West Boundary
      {
        dst(i,j).C = 0.0;
        dst(i,j).E  = src(i,j+1).W;
        dst(i,j).SE = src(i-1,j+1).NW;
        dst(i,j).NE = src(i+1,j+1).SW;
      }
      else if(ROWSIZE-1 == i) //North Boundary
      {
        dst(i,j).C = 0.0;
        dst(i,j).S  = src(i-1,j).N;
        dst(i,j).SE = src(i-1,j+1).NW;
        dst(i,j).SW = src(i-1,j-1).NE;
      }
      else if (COLSIZE -1 == j) //East Boundary
      {
        dst(i,j).C = 0.0;
        dst(i,j).W  = src(i,j-1).E;
        dst(i,j).NW = src(i+1,j-1).SE;
        dst(i,j).SW = src(i-1,j-1).NE;
      }
      else
      {
        if(ACCELERATION && ROWSIZE-2 == i){
          ux = 0.1; uy = 0.0; rho = 1.0;
        }
        else{
          ux = src(i,j-1).E + src(i-1,j-1).NE + src(i+1,j-1).SE;
          uy = src(i-1,j).N + src(i-1,j+1).NW;
          rho =   ux + uy + src(i,j).C + src(i,j+1).W + src(i+1,j+1).SW + src(i+1,j).S ;
          rhoinv = 1/rho;
          ux = (ux - src(i,j+1).W - src(i-1,j+1).NW - src(i+1,j+1).SW) * rhoinv;
          uy = (uy + src(i-1,j-1).NE - src(i+1,j).S - src(i+1,j-1).SE - src(i+1,j+1).SW ) * rhoinv;
        }
        u_sqr = (1.5 *( ux*ux + uy*uy));
        w_0 = OMEGA * W_0 * rho;
        w_1 = OMEGA * W_1 * rho;
        w_2 = OMEGA * W_2 * rho;
        dst(i,j).C = (src(i,j).C *  (1.0 - OMEGA)) +  w_0 * (1.0 - u_sqr);
        dst(i,j).E = (src(i,j-1).E * (1.0 - OMEGA)) + w_1* ( 1.0 - u_sqr + 3.0*ux + (4.5 * SQUAR(ux) ) );
        dst(i,j).W = (src(i,j+1).W * (1.0 - OMEGA)) + w_1* ( 1.0 - u_sqr - 3.0*ux + (4.5 * SQUAR(ux) ) );
        dst(i,j).N = (src(i-1,j).N * (1.0 - OMEGA)) + w_1* ( 1.0 - u_sqr + 3.0*uy + (4.5 * SQUAR(uy) ) );
        dst(i,j).S = (src(i+1,j).S * (1.0 - OMEGA)) + w_1* ( 1.0 - u_sqr - 3.0*uy + (4.5 * SQUAR(uy) ) );
    dst(i,j).NE = src(i-1,j-1).NE * (1.0 - OMEGA) + w_2* ( 1.0 - u_sqr + 3.0*(ux+uy) + (4.5 * SQUAR(ux+uy)));
    dst(i,j).NW = src(i-1,j+1).NW * (1.0 - OMEGA) + w_2* ( 1.0 -
        u_sqr + 3.0*(-ux+uy) + (4.5 * SQUAR(-ux+uy)));
    dst(i,j).SE = src(i+1,j-1).SE * (1.0 - OMEGA) + w_2* ( 1.0 -
        u_sqr + 3.0*(ux-uy) + (4.5 * SQUAR(ux-uy)));
    dst(i,j).SW = src(i+1,j+1).SW * (1.0 - OMEGA) + w_2* ( 1.0 -
        u_sqr + 3.0*(-ux-uy) + (4.5 * SQUAR(-ux-uy)));
      }
    }
  }
}
```

Listing B.2: Lattice Boltzmann Kernel in 2D without boundary conditions

```
void LBMKernel2D_withoutBnd(Cell* src, Cell* dst, int x0, int x1
   , int y0, int y1, int t0)
{
  double  rho = 0.0, ux = 0.0, uy = 0.0, u_sqr = 0.0, w_0 = 0.0,
      w_1 = 0.0, w_2 = 0.0, rhoinv = 0.0;
  int i = 0, j = 0;

  for(i=x0;i<x1;++i)
  {
#pragma ivdep
#pragma vector always
    for(j=y0;j<y1;++j)
    {
      if(ACCELERATION && ROWSIZE-2 == i){
        ux = 0.1; uy = 0.0; rho = 1.0;
      }
      else{
        ux = src(i,j-1).E + src(i-1,j-1).NE + src(i+1,j-1).SE;
        uy = src(i-1,j).N + src(i-1,j+1).NW;
        rho =   ux + uy + src(i,j).C + src(i,j+1).W + src(i+1,j
           +1).SW + src(i+1,j).S ;
        rhoinv = 1/rho;
        ux = (ux - src(i,j+1).W - src(i-1,j+1).NW - src(i+1,j+1)
           .SW) * rhoinv;
        uy = (uy + src(i-1,j-1).NE - src(i+1,j).S - src(i+1,j-1)
           .SE - src(i+1,j+1).SW ) * rhoinv;
      }
      u_sqr = (1.5 *( ux*ux + uy*uy));
      w_0 = OMEGA * W_0 * rho;
      w_1 = OMEGA * W_1 * rho;
      w_2 = OMEGA * W_2 * rho;
      dst(i,j).C = (src(i,j).C *  (1.0 - OMEGA)) +  w_0 * ( 1.0
         - u_sqr);
      dst(i,j).E = (src(i,j-1).E * (1.0 - OMEGA)) + w_1* ( 1.0 -
         u_sqr + 3.0*ux + (4.5 * SQUAR(ux) ) );
      dst(i,j).W = (src(i,j+1).W * (1.0 - OMEGA)) + w_1* ( 1.0 -
         u_sqr - 3.0*ux + (4.5 * SQUAR(ux) ) );
      dst(i,j).N = (src(i-1,j).N * (1.0 - OMEGA)) + w_1* ( 1.0 -
            u_sqr + 3.0*uy + (4.5 * SQUAR(uy) ) );
        dst(i,j).S = (src(i+1,j).S * (1.0 - OMEGA)) + w_1* ( 1.0 -
            u_sqr - 3.0*uy + (4.5 * SQUAR(uy) ) );
        dst(i,j).NE = src(i-1,j-1).NE * (1.0 - OMEGA) + w_2* ( 1.0
            - u_sqr + 3.0*(ux+uy) + (4.5 * SQUAR(ux+uy)));
        dst(i,j).NW = src(i-1,j+1).NW * (1.0 - OMEGA) + w_2* ( 1.0
            - u_sqr + 3.0*(-ux+uy) + (4.5 * SQUAR(-ux+uy)));
        dst(i,j).SE = src(i+1,j-1).SE * (1.0 - OMEGA) + w_2* ( 1.0
            - u_sqr + 3.0*(ux-uy) + (4.5 * SQUAR(ux-uy)));
        dst(i,j).SW = src(i+1,j+1).SW * (1.0 - OMEGA) + w_2* ( 1.0
            - u_sqr + 3.0*(-ux-uy) + (4.5 * SQUAR(-ux-uy)));
      }
    }
}
```

Listing B.3: Lattice Boltzmann Kernel in 3D.

```
void LBMKernel3D(Cell* src, Cell* dst, int x0, int x1, int y0,
     int y1, int z0, int z1)
{
  double  rho = 0.0, ux = 0.0, uy = 0.0, uz = 0.0, u_sqr = 0.0,
     w_0 = 0.0, w_1 = 0.0, w_2 = 0.0, rhoinv = 0.0;
  int i = 0, j = 0, k = 0;

#if !defined FRIGO && !defined FRIGOND
  #ifdef OPENMP
  #pragma omp parallel for private(rho, ux, uy, uz, u_sqr, w_0,
     w_1, w_2, rhoinv, k, i, j)
  #endif
#endif
  for(k=z0;k<z1;k++)
  {
    for(i=x0;i<x1;++i)
    {
      for(j=y0;j<y1;++j)
      {
        if(0 == k)  // Bottom Boundary
        {
          dst(k,i,j).C = 0.0;
          dst(k,i,j).T = src(k+1,i,j).B;
          dst(k,i,j).TN = src(k+1,i+1,j).BS;
          dst(k,i,j).TS = src(k+1,i-1,j).BN;
          dst(k,i,j).TE = src(k+1,i,j+1).BW;
          dst(k,i,j).TW = src(k+1,i,j-1).BE;
        }
        else if (LAYSIZE -1 == k) //Top Boundary
        {
          dst(k,i,j).C = 0.0;
          dst(k,i,j).B  = src(k-1,i,j).T;
          dst(k,i,j).BN  = src(k-1,i+1,j).TS;
          dst(k,i,j).BS  = src(k-1,i-1,j).TN;
          dst(k,i,j).BE  = src(k-1,i,j+1).TW;
          dst(k,i,j).BW  = src(k-1,i,j-1).TE;
        }
        else if(0 == i) //  South Boundary
        {
          dst(k,i,j).C = 0.0;
          dst(k,i,j).N  = src(k,i+1,j).S;
          dst(k,i,j).NE = src(k,i+1,j+1).SW;
          dst(k,i,j).NW = src(k,i+1,j-1).SE;
          dst(k,i,j).TN = src(k+1,i+1,j).BS;
          dst(k,i,j).BN = src(k-1,i+1,j).TS;
        }
        else if(ROWSIZE-1 == i) //North Boundary
        {
          dst(k,i,j).C = 0.0;
          dst(k,i,j).S  = src(k,i-1,j).N;
          dst(k,i,j).SE = src(k,i-1,j+1).NW;
          dst(k,i,j).SW = src(k,i-1,j-1).NE;
          dst(k,i,j).TS = src(k+1,i-1,j).BN;
          dst(k,i,j).BS = src(k-1,i-1,j).TN;
        }
        else if(0 == j) //West Boundary
        {
          dst(k,i,j).C = 0.0;
          dst(k,i,j).E  = src(k,i,j+1).W;
          dst(k,i,j).SE = src(k,i-1,j+1).NW;
          dst(k,i,j).NE = src(k,i+1,j+1).SW;
          dst(k,i,j).TE = src(k+1,i,j+1).BW;
          dst(k,i,j).BE = src(k-1,i,j+1).TW;
        }
        else if (COLSIZE -1 == j) //East Boundary
        {
          dst(k,i,j).C = 0.0;
          dst(k,i,j).W  = src(k,i,j-1).E;
          dst(k,i,j).NW = src(k,i+1,j-1).SE;
          dst(k,i,j).SW = src(k,i-1,j-1).NE;
          dst(k,i,j).TW = src(k+1,i,j-1).BE;
          dst(k,i,j).BW = src(k-1,i,j-1).TE;
        }
        else
        {
          if(ACCELERATION && ROWSIZE-2 == i){
            ux = 0.1 * 3.0; uy = 0.0;
            uz = 0.0; rho = 1.0;
          }
          else{
            ux = src(k,i,j-1).E + src(k,i-1,j-1).NE + src(k,i+1,
               j-1).SE + src(k-1,i,j-1).TE + src(k+1,i,j-1).BE;
```

```
                uy = src(k,i-1,j).N + src(k,i-1,j+1).NW + src(k-1,i
                   -1,j).TN + src(k+1,i-1,j).BN;
                uz = src(k-1,i,j).T + src(k-1,i+1,j).TS + src(k-1,i,
                   j+1).TW;
                rho =   ux + uy + uz + src(k,i,j).C + src(k,i,j+1).W
                    + src(k,i+1,j+1).SW + src(k,i+1,j).S + src(k+1,i
                   ,j).B + src(k+1,i+1,j).BS + src(k+1,i,j+1).BW;

                rhoinv = 3/rho;
                ux = (ux - src(k,i,j+1).W - src(k,i-1,j+1).NW - src(
                   k,i+1,j+1).SW - src(k-1,i,j+1).TW - src(k+1,i,j
                   +1).BW) * rhoinv;

                uy = (uy + src(k,i-1,j-1).NE - src(k,i+1,j).S - src(
                   k,i+1,j-1).SE - src(k,i+1,j+1).SW - src(k-1,i+1,j
                   ).TS - src(k+1,i+1,j).BS) * rhoinv;

                uz = (uz + src(k-1,i-1,j).TN + src(k-1,i,j-1).TE -
                   src(k+1,i,j).B - src(k+1,i-1,j).BN - src(k+1,i,j
                   -1).BE - src(k+1,i+1,j).BS - src(k+1,i,j+1).BW) *
                    rhoinv;
              }
              u_sqr = 1.0 - (0.1666666666666667 *( ux*ux + uy*uy +
                 uz*uz));
              w_0 = OMEGA * W_0 * rho;
              w_1 = OMEGA * W_1 * rho;
              w_2 = OMEGA * W_2 * rho;
              dst(k,i,j).C = (src(k,i,j).C *  (1.0 - OMEGA)) +  w_0
                 * (u_sqr);
              dst(k,i,j).E = (src(k,i,j-1).E * (1.0 - OMEGA)) + w_1*
                  ( u_sqr + ux + (0.5 * SQUAR(ux) ));
              dst(k,i,j).W = (src(k,i,j+1).W * (1.0 - OMEGA)) + w_1*
                  ( u_sqr - ux + (0.5 * SQUAR(ux) ));
              dst(k,i,j).N = (src(k,i-1,j).N * (1.0 - OMEGA)) + w_1*
                  ( u_sqr + uy + (0.5 * SQUAR(uy) ));
              dst(k,i,j).S = (src(k,i+1,j).S * (1.0 - OMEGA)) + w_1*
                  ( u_sqr - uy + (0.5 * SQUAR(uy) ));
              dst(k,i,j).T = (src(k-1,i,j).T * (1.0 - OMEGA)) + w_1*
                  ( u_sqr + uz + (0.5 * SQUAR(uz) ));
              dst(k,i,j).B = (src(k+1,i,j).B * (1.0 - OMEGA)) + w_1*
                  ( u_sqr - uz + (0.5 * SQUAR(uz) ));
          dst(k,i,j).NE = src(k,i-1,j-1).NE * (1.0 - OMEGA) + w_2* (
              u_sqr + (ux+uy) + (0.5 * SQUAR(ux+uy)));
          dst(k,i,j).NW = src(k,i-1,j+1).NW * (1.0 - OMEGA) + w_2*(
             u_sqr + (-ux+uy) + (0.5 *SQUAR(-ux+uy)));
          dst(k,i,j).SE = src(k,i+1,j-1).SE * (1.0 - OMEGA) + w_2* (
              u_sqr + (ux-uy) + (0.5 * SQUAR(ux-uy)));
          dst(k,i,j).SW = src(k,i+1,j+1).SW * (1.0 - OMEGA) + w_2*(
             u_sqr + (-ux-uy) + (0.5 *SQUAR(-ux-uy)));
```

```
      dst(k,i,j).TN = src(k-1,i-1,j).TN * (1.0 - OMEGA) + w_2* (
          u_sqr + (uz+uy) + (0.5 * SQUAR(uz+uy)));
      dst(k,i,j).TS = src(k-1,i+1,j).TS * (1.0 - OMEGA) + w_2* (
          u_sqr + (uz-uy) + (0.5 * SQUAR(uz-uy)));
      dst(k,i,j).TW = src(k-1,i,j+1).TW * (1.0 - OMEGA) + w_2* (
          u_sqr + (uz-ux) + (0.5 * SQUAR(uz-ux)));
      dst(k,i,j).TE = src(k-1,i,j-1).TE * (1.0 - OMEGA) + w_2* (
          u_sqr + (uz+ux) + (0.5 * SQUAR(uz+ux)));
      dst(k,i,j).BN = src(k+1,i-1,j).BN * (1.0 - OMEGA) + w_2* (
          u_sqr + (uy-uz) + (0.5 * SQUAR(uy-uz)));
      dst(k,i,j).BS = src(k+1,i+1,j).BS * (1.0 - OMEGA) + w_2*(
         u_sqr + (-uy-uz) + (0.5 *SQUAR(-uy-uz)));
      dst(k,i,j).BW = src(k+1,i,j+1).BW * (1.0 - OMEGA) + w_2*(
         u_sqr + (-ux-uz) + (0.5 *SQUAR(-ux-uz)));
      dst(k,i,j).BE = src(k+1,i,j-1).BE * (1.0 - OMEGA) + w_2* (
          u_sqr + (ux-uz) + (0.5 * SQUAR(ux-uz)));
        }
      }
    }
  }
}
```

Listing B.4: Lattice Boltzmann Kernel in 3D without boundary conditions.

```
void LBMKernel3D_withoutBnd(Cell* src, Cell* dst, int x0, int x1
    , int y0, int y1, int z0, int z1)
{
  double  rho = 0.0, ux = 0.0, uy = 0.0, uz = 0.0, u_sqr = 0.0,
     w_0 = 0.0,  w_1 = 0.0, w_2 = 0.0, rhoinv = 0.0;
  int i = 0, j = 0, k = 0;

  for(k=z0;k<z1;k++)
  {
    for(i=x0;i<x1;++i)
    {
      if(ACCELERATION && ROWSIZE-2 == i)
      {
        ux = 0.1 * 3.0; uy = 0.0;
        uz = 0.0; rho = 1.0;
        u_sqr = 1.0 - (0.166666666666667 *( ux*ux + uy*uy + uz*
           uz));
        w_0 = OMEGA * W_0 * rho;
        w_1 = OMEGA * W_1 * rho;
        w_2 = OMEGA * W_2 * rho;

        #pragma ivdep
        #pragma vector always
        for(j=y0;j<y1;++j)
        {
            dst(k,i,j).C = (src(k,i,j).C *  (1.0 - OMEGA)) +
                  w_0 * (u_sqr);
            dst(k,i,j).E = (src(k,i,j-1).E * (1.0 - OMEGA)) + w_1*
                ( u_sqr + ux + (0.5 * SQUAR(ux) ));
            dst(k,i,j).W = (src(k,i,j+1).W * (1.0 - OMEGA)) + w_1*
                ( u_sqr - ux + (0.5 * SQUAR(ux) ));
            dst(k,i,j).N = (src(k,i-1,j).N * (1.0 - OMEGA)) + w_1*
                ( u_sqr + uy + (0.5 * SQUAR(uy) ));
            dst(k,i,j).S = (src(k,i+1,j).S * (1.0 - OMEGA)) + w_1*
                ( u_sqr - uy + (0.5 * SQUAR(uy) ));
            dst(k,i,j).T = (src(k-1,i,j).T * (1.0 - OMEGA)) + w_1*
                ( u_sqr + uz + (0.5 * SQUAR(uz) ));
            dst(k,i,j).B = (src(k+1,i,j).B * (1.0 - OMEGA)) + w_1*
                ( u_sqr - uz + (0.5 * SQUAR(uz) ));
          dst(k,i,j).NE = src(k,i-1,j-1).NE * (1.0 - OMEGA) + w_2*
              ( u_sqr + (ux+uy) + (0.5 * SQUAR(ux+uy)));
          dst(k,i,j).NW = src(k,i-1,j+1).NW * (1.0 - OMEGA) + w_2
             *( u_sqr + (-ux+uy) + (0.5 *SQUAR(-ux+uy)));
          dst(k,i,j).SE = src(k,i+1,j-1).SE * (1.0 - OMEGA) + w_2*
              ( u_sqr + (ux-uy) + (0.5 * SQUAR(ux-uy)));
          dst(k,i,j).SW = src(k,i+1,j+1).SW * (1.0 - OMEGA) + w_2
             *( u_sqr + (-ux-uy) + (0.5 *SQUAR(-ux-uy)));
          dst(k,i,j).TN = src(k-1,i-1,j).TN * (1.0 - OMEGA) + w_2*
              ( u_sqr + (uz+uy) + (0.5 * SQUAR(uz+uy)));
          dst(k,i,j).TS = src(k-1,i+1,j).TS * (1.0 - OMEGA) + w_2*
              ( u_sqr + (uz-uy) + (0.5 * SQUAR(uz-uy)));
          dst(k,i,j).TW = src(k-1,i,j+1).TW * (1.0 - OMEGA) + w_2*
              ( u_sqr + (uz-ux) + (0.5 * SQUAR(uz-ux)));
          dst(k,i,j).TE = src(k-1,i,j-1).TE * (1.0 - OMEGA) + w_2*
              ( u_sqr + (uz+ux) + (0.5 * SQUAR(uz+ux)));
          dst(k,i,j).BN = src(k+1,i-1,j).BN * (1.0 - OMEGA) + w_2*
              ( u_sqr + (uy-uz) + (0.5 * SQUAR(uy-uz)));
          dst(k,i,j).BS = src(k+1,i+1,j).BS * (1.0 - OMEGA) + w_2
             *( u_sqr + (-uy-uz) + (0.5 *SQUAR(-uy-uz)));
          dst(k,i,j).BW = src(k+1,i,j+1).BW * (1.0 - OMEGA) + w_2
             *( u_sqr + (-ux-uz) + (0.5 *SQUAR(-ux-uz)));
          dst(k,i,j).BE = src(k+1,i,j-1).BE * (1.0 - OMEGA) + w_2*
              ( u_sqr + (ux-uz) + (0.5 * SQUAR(ux-uz)));
          } //end of for j
        }
        else
        {
          #pragma ivdep
          #pragma vector always
          for(j=y0;j<y1;++j)
          {
          ux = src(k,i,j-1).E + src(k,i-1,j-1).NE + src(k,i+1,j-1)
             .SE + src(k-1,i,j-1).TE + src(k+1,i,j-1).BE;
          uy = src(k,i-1,j).N + src(k,i-1,j+1).NW + src(k-1,i-1,j)
             .TN + src(k+1,i-1,j).BN;
          uz = src(k-1,i,j).T + src(k-1,i+1,j).TS + src(k-1,i,j+1)
             .TW;
          rho =   ux + uy + uz + src(k,i,j).C + src(k,i,j+1).W +
             src(k,i+1,j+1).SW + src(k,i+1,j).S + src(k+1,i,j).B +
              src(k+1,i+1,j).BS + src(k+1,i,j+1).BW;

          rhoinv = 3/rho;
          ux = (ux - src(k,i,j+1).W - src(k,i-1,j+1).NW - src(k,i
             +1,j+1).SW - src(k-1,i,j+1).TW - src(k+1,i,j+1).BW) *
              rhoinv;

          uy = (uy + src(k,i-1,j-1).NE - src(k,i+1,j).S - src(k,i
             +1,j-1).SE - src(k,i+1,j+1).SW - src(k-1,i+1,j).TS -
             src(k+1,i+1,j).BS) * rhoinv;

          uz = (uz + src(k-1,i-1,j).TN + src(k-1,i,j-1).TE - src(k
             +1,i,j).B - src(k+1,i-1,j).BN - src(k+1,i,j-1).BE -
             src(k+1,i+1,j).BS - src(k+1,i,j+1).BW) * rhoinv;

          u_sqr = 1.0 - (0.1666666666666667  *( ux*ux + uy*uy + uz
             *uz));
          w_0 = OMEGA * W_0 * rho;
          w_1 = OMEGA * W_1 * rho;
          w_2 = OMEGA * W_2 * rho;
          dst(k,i,j).C = (src(k,i,j).C *  (1.0 - OMEGA)) +  w_0 *
             (u_sqr);
          dst(k,i,j).E = (src(k,i,j-1).E * (1.0 - OMEGA)) + w_1* (
              u_sqr + ux + (0.5 * SQUAR(ux) ));
          dst(k,i,j).W = (src(k,i,j+1).W * (1.0 - OMEGA)) + w_1* (
              u_sqr - ux + (0.5 * SQUAR(ux) ));
          dst(k,i,j).N = (src(k,i-1,j).N * (1.0 - OMEGA)) + w_1* (
              u_sqr + uy + (0.5 * SQUAR(uy) ));
          dst(k,i,j).S = (src(k,i+1,j).S * (1.0 - OMEGA)) + w_1* (
              u_sqr - uy + (0.5 * SQUAR(uy) ));
          dst(k,i,j).T = (src(k-1,i,j).T * (1.0 - OMEGA)) + w_1* (
              u_sqr + uz + (0.5 * SQUAR(uz) ));
          dst(k,i,j).B = (src(k+1,i,j).B * (1.0 - OMEGA)) + w_1* (
              u_sqr - uz + (0.5 * SQUAR(uz) ));
        dst(k,i,j).NE = src(k,i-1,j-1).NE * (1.0 - OMEGA) + w_2* (
            u_sqr + (ux+uy) + (0.5 * SQUAR(ux+uy)));
        dst(k,i,j).SW = src(k,i+1,j+1).SW * (1.0 - OMEGA) + w_2*(
           u_sqr - (ux+uy) + (0.5 *SQUAR(ux+uy)));
        dst(k,i,j).NW = src(k,i-1,j+1).NW * (1.0 - OMEGA) + w_2*(
           u_sqr - (ux-uy) + (0.5 *SQUAR(-ux+uy)));
        dst(k,i,j).SE = src(k,i+1,j-1).SE * (1.0 - OMEGA) + w_2*(
           u_sqr + (ux-uy) + (0.5 * SQUAR(ux-uy)));
        dst(k,i,j).TN = src(k-1,i-1,j).TN * (1.0 - OMEGA) + w_2*(
           u_sqr + (uz+uy) + (0.5 * SQUAR(uz+uy)));
        dst(k,i,j).BS = src(k+1,i+1,j).BS * (1.0 - OMEGA) + w_2*(
            u_sqr - (uz+uy) + (0.5 *SQUAR(uy+uz)));
        dst(k,i,j).TS = src(k-1,i+1,j).TS * (1.0 - OMEGA) + w_2* (
            u_sqr + (uz-uy) + (0.5 * SQUAR(uz-uy)));
        dst(k,i,j).BN = src(k+1,i-1,j).BN * (1.0 - OMEGA) + w_2* (
            u_sqr - (uz-uy) + (0.5 * SQUAR(uy-uz)));
        dst(k,i,j).TW = src(k-1,i,j+1).TW * (1.0 - OMEGA) + w_2* (
             u_sqr + (uz-ux) + (0.5 * SQUAR(uz-ux)));
        dst(k,i,j).BE = src(k+1,i,j-1).BE * (1.0 - OMEGA) + w_2* (
             u_sqr - (uz-ux) + (0.5 * SQUAR(ux-uz)));
        dst(k,i,j).TE = src(k-1,i,j-1).TE * (1.0 - OMEGA) + w_2* (
             u_sqr + (uz+ux) + (0.5 * SQUAR(uz+ux)));
        dst(k,i,j).BW = src(k+1,i,j+1).BW * (1.0 - OMEGA) + w_2*(
            u_sqr - (uz+ux) + (0.5 *SQUAR(ux+uz)));
      } // end of for j
    } // end of else
  } // end of for i
} // end of for k
}
```

Listing B.5: COLBA : Prefetching inside COLBA: Complete implementation.

```
void walk3D_withPrefetch(int t0, int t1, int x0, int dx0, int x1
   , int dx1, int y0, int dy0, int y1, int dy1, int z0, int dz0,
    int z1, int dz1)
{
  int dt = t1 - t0;
  if(dt == 0 )
    return;
  if(dt == 1){
    if(firstCheckFlag == 'T')
    {
      firstCheckFlag = 'F';
      pret0 = t0; pret1 = t1;
      prex0 = x0; prex1 = x1;
      prey0 = y0; prey1 = y1;
      prez0 = z0; prez1 = z1;
    }
    else
    {
      if((pret0 & 1) != 0) //Odd time step
      {
        if(0 == prex0 || 0 == prey0 || 0 == prez0 || ROWSIZE ==
           prex1 || COLSIZE == prey1 || LAYSIZE == prez1)
        {
          LBMKernel3D( dst3D, src3D, prex0, prex1, prey0, prey1,
               prez0, prez1);
        }
        else
          LBMKernel3D_withoutBnd( dst3D, src3D, prex0, prex1,
             prey0, prey1, prez0, prez1);
      }
      else  //Even time step
      {
        if(0 == prex0 || 0 == prey0 || 0 == prez0 || ROWSIZE ==
           prex1 || COLSIZE == prey1 || LAYSIZE == prez1)
        {
          LBMKernel3D( src3D, dst3D, prex0, prex1, prey0, prey1,
               prez0, prez1);
        }
        else
          LBMKernel3D_withoutBnd( src3D, dst3D, prex0, prex1,
             prey0, prey1, prez0, prez1);
      }
      pret0 = t0; pret1 = t1;
      prex0 = x0; prex1 = x1;
      prey0 = y0; prey1 = y1;
      prez0 = z0; prez1 = z1;
      prefetchNextBlock(x0, x1, y0, y1, z0, z1);
    }
  }
  else {
    if (SPC * (x1 - x0) + (dx1 - dx0) * dt >= 4  * dt) {
      int xm = (int)(( 2*(x1+x0) + ((2+dx0+dx1)*dt) )>>2);
      walk3D_withPrefetch(t0, t1, x0, dx0, xm, -1, y0, dy0, y1,
         dy1, z0, dz0, z1, dz1);
      walk3D_withPrefetch(t0, t1, xm, -1, x1, dx1, y0, dy0, y1,
         dy1, z0, dz0, z1, dz1);
    }
    else if (SPC * (y1 - y0) + (dy1 - dy0) * dt >= 4  * dt) {
      int ym = (int)((2*(y0+y1) + ((2+dy0+dy1)*dt))>>2);
      walk3D_withPrefetch(t0, t1, x0, dx0, x1, dx1, y0, dy0, ym,
          -1, z0, dz0, z1, dz1);
      walk3D_withPrefetch(t0, t1, x0, dx0, x1, dx1, ym, -1, y1,
         dy1, z0, dz0, z1, dz1);
    }
    else if(SPC * (z1 - z0) + (dz1 - dz0) * dt >= 4  * dt) {
      int zm = (int)((2*(z0+z1) + ((2+dz0+dz1)*dt))>>2);
      walk3D_withPrefetch(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1,
          dy1, z0, dz0, zm, -1);
      walk3D_withPrefetch(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1,
          dy1, zm, -1, z1, dz1);
    }
    else {
        int s = dt/2;
        walk3D_withPrefetch( t0, t0+s, x0, dx0, x1, dx1, y0, dy0
           , y1, dy1, z0, dz0, z1, dz1);
        walk3D_withPrefetch( t0+s, t1,  (x0+(dx0*s)), dx0, (x1+(
           dx1*s)), dx1, (y0+(dy0*s)), dy0, (y1+(dy1*s)), dy1, (
           z0+(dz0*s)), dz0, (z1+(dz1*s)), dz1);
    }
  }
}
```

Listing B.6: COLBA : The complete implementation of parallelWalk() function.

```
void parallelWalk(int t0, int t1, int x0, int dx0, int x1, int
    dx1, int y0, int dy0, int y1, int dy1, int z0, int dz0, int
    z1, int dz1)
{
  if( dy0 == 0 || dy1 == 0)
  {
    return;
  }
  else //(dy0 == -1 || dy1 == -1)
  {
    if((t1 != totalTs))// && (t1 !> totalTs))
    {
      if(y0 == DT && y1 == 2*DT)
      {
        #pragma intel omp taskq
        {
          #pragma intel omp task
          {
              //check bottom parallelogram.
            if(t0 >= DT)
            {
              if(omp_test_lock(&syncTable[(((t0-DT)/DT) * stCol)
                  + ((y1+DT)/DT) ]) )
              {
                walk3D(t0, t1,  x0, dx0, x1, dx1, y0+DT, -1, y1+
                    DT, -1, z0, dz0, z1, dz1);
              }
              else{
                omp_set_lock(&syncTable[(((t0-DT)/DT) * stCol) +
                    ((y1+DT)/DT) ]);
                walk3D(t0, t1,  x0, dx0, x1, dx1, y0+DT, -1, y1+
                    DT, -1, z0, dz0, z1, dz1);
              }
            }
            else
              walk3D(t0, t1,  x0, dx0, x1, dx1, y0+DT, -1, y1+DT
                  , -1, z0, dz0, z1, dz1);
            //unlock the parallelogram.
            omp_unset_lock(&syncTable[((t0/DT) * stCol) + ((y0+
                DT)/DT) ]);
            parallelWalk(t0, t1,  x0, dx0, x1, dx1,     y0+DT,
                -1, y1+DT, -1,   z0, dz0, z1, dz1);
          }
          #pragma intel omp task
        {
          walk3D(t1, t1+DT, x0, dx0, x1, dx1,     y0-DT, 0, y1-DT, -1,     z0, dz0, z1, dz1);
          //unlock the triangle.
          omp_unset_lock(&syncTable[((t1/DT) * stCol) + ((y0-DT)/DT) ]);
          parallelWalk(t1, t1+DT, x0, dx0, x1, dx1,     y0-DT, 0, y1-DT, -1,     z0, dz0, z1, dz1);
        }
      } // taskq

      return;
    } //if
    else if(y0 == 2*DT && y1 == 3*DT)
    {

      #pragma intel omp taskq
      {
        #pragma intel omp task
        {
            //check bottom parallelogram.
          if(t0 >= DT)
          {
            if(omp_test_lock(&syncTable[(((t0-DT)/DT) * stCol) + ((y1+DT)/DT) ]) )
            {
              walk3D(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+DT, -1,   z0, dz0, z1, dz1);
            }
            else{
              omp_set_lock(&syncTable[(((t0-DT)/DT) * stCol) + ((y1+DT)/DT) ]);
              walk3D(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+DT, -1,   z0, dz0, z1, dz1);
            }
          }
          else
            walk3D(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+DT, -1,   z0, dz0, z1, dz1);
          //unlock the parallelogram.
          omp_unset_lock(&syncTable[((t0/DT) * stCol) + ((y0+DT)/DT) ]);
          parallelWalk(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+DT, -1,   z0, dz0, z1, dz1);
        }
        #pragma intel omp task
        {
          if(omp_test_lock(&syncTable[((t1/DT) * stCol) + ((y0-(2*DT))/DT) ]) )
        walk3D(t1, t1+DT, x0, dx0, x1, dx1,     y0-DT, -1, y1-DT, -1,   z0, dz0, z1, dz1);
       else
       {
        omp_set_lock(&syncTable[((t1/DT) * stCol) + ((y0-(2*DT))/DT) ]);
        walk3D(t1, t1+DT, x0, dx0, x1, dx1,     y0-DT, -1, y1-DT, -1,   z0, dz0, z1, dz1);
       }
       //unlock the parallelogram.
       omp_unset_lock(&syncTable[((t1/DT) * stCol) + ((y0-DT)/DT) ]);
       parallelWalk(t1, t1+DT, x0, dx0, x1, dx1,     y0-DT, -1, y1-DT, -1,   z0, dz0, z1, dz1);
      }
     } // taskq

     return;
    } //else if
    else if(y1 == COLSIZE)    //solves triangles in last columns.
    {
     walk3D(t0, t1,    x0, dx0, x1, dx1,     y0+DT, -1, y1, 0,   z0, dz0, z1, dz1);
     // unlock the triangle.
     omp_unset_lock(&syncTable[((t0/DT) * stCol) + ((y0+DT)/DT) ]);
     return;
    }
    else{
     // solves the central domain.
     // check the bottom region; it should be unlock.
     if(t0 >= DT)
     {
      if(omp_test_lock(&syncTable[(((t0-DT)/DT) * stCol) + ((y1+DT)/DT) ]) )
      {
       walk3D(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+DT, -1,   z0, dz0, z1, dz1);
      }
      else{
       omp_set_lock(&syncTable[(((t0-DT)/DT) * stCol) + ((y1+DT)/DT) ]);
       walk3D(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+DT, -1,   z0, dz0, z1, dz1);
      }
     }
     else
      walk3D(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+DT
                 , -1,   z0, dz0, z1, dz1);

        //unlock the parallelogram.
        omp_unset_lock(&syncTable[((t0/DT) * stCol) + ((y0+DT)/
           DT) ]);
        parallelWalk(t0, t1,  x0, dx0, x1, dx1,     y0+DT, dy0,
           y1+DT, dy1,   z0, dz0, z1, dz1);
        return;
      }
    }
    else{
        if(y1 == COLSIZE) //solves right hand side top most
           triangle.
        {
          walk3D(t0, t1,    x0, dx0, x1, dx1,     y0+DT, -1, y1,
              0,   z0, dz0, z1, dz1);
          // unlock triangle. But it won't affect because it is
             the last region.
          omp_unset_lock(&syncTable[((t0/DT) * stCol) + ((y0+DT)
             /DT) ]);
          return;
        }
        else    // solves the top most row.
        {
          // check the bottom region; it should be unlock.
          if(omp_test_lock(&syncTable[(((t0-DT)/DT) * stCol) +
             ((y1+DT)/DT) ]) )
          {
            walk3D(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+
               DT, -1,   z0, dz0, z1, dz1);
          }
          else{
            omp_set_lock(&syncTable[(((t0-DT)/DT) * stCol) + ((
               y1+DT)/DT) ]);
            walk3D(t0, t1,  x0, dx0, x1, dx1,     y0+DT, -1, y1+
               DT, -1,   z0, dz0, z1, dz1);
          }
          // unlock the parallelogram; it won't affect.
          omp_unset_lock(&syncTable[((t0/DT) * stCol) + ((y0+DT)
             /DT) ]);
          parallelWalk(t0, t1, x0, dx0, x1, dx1,    y0+DT, dy0,
             y1+DT, dy1,   z0, dz0, z1, dz1);
          return;
        }
    }//else
    return;
  } //else
  return;
}
```

Appendix C

Compiling and Running the Program:

C.1 Makefile

It has 2 options for compilation. One with PAPI and one without PAPI. If 'PERFORMANCE_MEASUREMENT' flag is declared then the program will compile for PAPI.

C.1.1 Compilation Flags

FRIGO: To execute the program with the COLBA (for 2D or 3D). If commented out the program will execute the iterative version.

FRIGOND: To execute the program with the COLBA (for any (N-) dimension). If commented out, the program will execute the iterative version.

PAPI: To measure the performance with PAPI. If commented out the PAPI functionality will not be executed.

DEBUG: To print some tracing information and error messages.

OPENMP: For parallel processing. Parallelization is implemented for the COLBA as well as for the iterative algorithm.

COLBA (parallel): OPENMP + FRIGO.

Iterative (parallel): OPENMP.

C.1.2 #defines in Program

The following #defines declared inside program can be sets through Makefile.

1. -DCOLSIZE=$(colsize) -DROWSIZE=$(rowsize) -DLAYSIZE=$(laysize)

   The above 3 variables represent the domain size. If these variables are not specified (i.e. commented out) then by default program runs for a domain size of (100x100x100).

2. -DSPC It is the space cut deciding factor in COLBA. If commented out, the default value of 2.0 is used.

Note: Be sure that for PAPI and OPENMP the library paths are correctly specified. Use either FRIGO or FRIGOND for running the program with COLBA.

C.1.3 Compiler Flags and Executable:

Flags used with ICC Compiler (version 9.0):

- -Wall
- -O3
- -static
- -xN, -xP, -xT(with Intel compiler version 9.1)
- -fno-alias
- -openmp (for parallel execution)
- -opt_report_file ”report_opt.txt” (for generating optimization report)
- -opt_report_level ”max” (for generating optimization report)
- -g (along with -prof_genx and -prof_use)
- -prof_genx (profile guided optimization)
- -prof_use (profile guided optimization)

Flags used with GCC Compiler (Version 4.1):

- -Wall
- -O3
- -static
- -fargument-noalias-global
- -openmp (for parallel execution)

Name of executable: frigo3D.

Command line argument (Single processor): ./frigo3D [Number of time steps.]

Command line argument (Multi-processor):
./frigo3D [Number of time steps.] [Time blocking size]

C.2 Source Code Files

main.c: This file contains the main() function, which calls all functionality through the initialize() function. The domain size(x1, y1 and z1) is read from the Makefile. If it is not defined in the Makefile, then by default 100^3 is used. x1 represents No. of rows. y1 represents No. of columns. z1 represents No. of layers.

Cell.h: Declaration of 'struct Cell', 'struct Dimension', 'struct indexTable' and macros COLSIZE, ROWSIZE & LAYSIZE. (if not defined in the Makefile.)

frigo3D.h:

- Declaration of all functions in frigo3D.c, frigoND.c, frigoPar.c, frigoPrefetch.c & lbm3D.c
- Declaration of the SPC (if not defined in the Makefile) and the blocking factors. (X_BLOCK_SIZE ...)
- Declaration of the macros to access source and destination grid, as well as some #defines.

frigo3D.c: This file contains the implementation of the COLBA (walk3D())and the iterative implementation (testmain()). Both implementations are for the 3 dimensional case. The file contains some functions for debugging and for performance measurement using PAPI. The 2-way (iterativeBlocked()) blocking for iterative code is also implemented in this file.

frigoND.c: (The file is included at the end of frigo3D.c) This file contains Frigo's algorithm for 'N' dimensions. The walkND() takes its argument as a pointer to an array of dimensions. So there is no need to pass the individual dimensions in each direction. Therefore, the function has the flexibility to work with any dimension. (For 1D, 2D, 3D same function can be used.)

frigoPar.c: (The file is included at the end of frigo3D.c) This file contains the parallel implementation of COLBA.

frigoPrefetch.c: (The file is included at the end of frigo3D.c) This file contains the COLBA with a pre-fetching function.

lbm3D.c: (The file is included at the end of frigo3D.c) This file contains the implementation LBMKernel3D(). It also contains separate functions for boundary fluid cells and fluid (surrounded by fluid) cells.

fileWriter.h: Declaration of all functions in fileWriter.c .

fileWriter.c: Mainly, this file contains functions to write the values in a file (in text format). One function writes files in the OpenDx format for a visualization. Other functions write performance measuring values like L1, L2 cache misses or wall clock timings.

List of Figures

List of Tables

List of Algorithms

Listings

Bibliography

[1] Home page OpenDX. http://www.opendx.org/.

[2] Stefan Donath. On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures, 2004. Lehrstuhl für Systemsimulation, Universität Erlangen-Nürnberg.

[3] Matteo Frigo and Volker Strumpen. Cache Oblivious Stencil Computations. *IBM Austin Research Laboratory, 11501 Burnet Road, Austin, TX 78758*, 2005. Available at : http://www.fftw.org/ athena/papers/ics05.pdf.

[4] Klaus Iglberger. Cache Optimizations for the Lattice Boltzmann Method in 3D, 2003. Lehrstuhl für Systemsimulation, Universität Erlangen-Nürnberg.

[5] Intel. Intel C++ Compiler for Linux* Systems User's Guide.

[6] Renwei Mei, Wei Shyy, and Dazhi Yu. Lattice Boltzmann Method for 3D Flows with Curved Boundary. *Technical Report NASA/CR-2002-211657, ICASE, NASA Langley Research Center, Hampton, Virginia*, June 2002.

[7] Herald Prokop. Cache-Oblivious Algorithm. Master's thesis, Department of Electrical Engineering and Computer Science, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 1999. Available at : http://supertech.lcs.mit.edu/cilk/papers/index.html.

[8] Jens Wilke. Cache Optimizations for the Lattice Boltzmann Method in 2D, 2003. Lehrstuhl für Systemsimulation, Universität Erlangen-Nürnberg.

[9] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models.* Springer, 2000.