# Efficient Implementation of a 2D Lattice Boltzmann Solver Using Modern Parallelisation Techniques

Máté Tibor Szőke

School of Engineering
Computational Fluid Dynamics
Cranfield University

Supervisors:
Dr Irene Moulitsas
Dr László Könözsy

August 2014

## Abstract

This masters thesis discusses the parallelisation of a two dimensional lattice Boltzmann solver. A new, C programming language based solver was born from two former in-house codes. This C code was parallelised using the Partitioned Global Address Space programming model. Unified Parallel C, as an extension to the standard C language, offers new opportunities to the programmer compared to conventional approaches (Message Passing Interface, OpenMP). Two parallel codes were formed. The first relies on shared memory, while the second exploits data locality. It was found that the shared variables introduced significant latency, therefore this code performed poorly. The local variable based solver overcame the limitations of the conventional parallel programming models, and the speed-up gained was better than the ideal speed-up. As a final step, the serial and the better-performing parallel code were validated. It was found that the codes resolved the same physics, but underestimated the velocity magnitudes.

## Acknowledgements

Opposite to the scientific journals, a thesis shows a single author on its cover. This, however, does not mean that it includes only the work of a single person. Therefore, I would like to express my very great appreciation to all the supporters who helped my work.

I would like to offer my special thanks to *Dr. Irene Moulitsas* for her helpful assistance. Her door was always open to discuss all the questions that showed up during my work. I am particularly grateful for all her helpful advices. I would like to thank *Dr. László Könözsy* for his support.

Special thanks belong to *Mr. Tamás István Józsa*, my colleague, classmate and friend. Without his cooperation, the common initial work would have never been done within such a short time. Furthermore, I express my thank for his invaluable assistance and our discussions on the lattice Boltzmann method.

Last but not least, I would like to thank my parents for their support throughout this busy and exhausting year.

Máté Tibor Szőke

# Contents

# List of Figures

# List of Tables

# List of Symbols

## Latin Letters

| Symbol | Definition | Unit |
|---|---|---|
| $\mathbf{c}$ | Lattice velocity | [m s$^{-1}$] |
| $f$ | Distribution function | [−] |
| $f^{eq}$ | Distribution function in equilibrium | [−] |
| $\mathbf{F}$ | Force | [N] |
| $g$ | Gravitational acceleration | [m s$^{-2}$] |
| $H$ | Characteristic length | [m] |
| $\mathbf{i}, \mathbf{j}$ | $x$ and $y$ unit vectors | [−] |
| $k_B$ | Boltzmann constant | [J K$^{-1}$] |
| $m$ | Mass | [kg] |
| $p$ | Pressure | [Pa] |
| $\mathbf{r}$ | Radius | [m] |
| $\mathbf{r}$ | Radius, distance from a point | [m] |
| $R$ | Residuals | [SI standard] |
| $S$ | Sum of variables | [SI standard] |
| $\mathbf{S}$ | Relaxation matrix | [SI standard] |
| $SU$ | Speed-up | [−] |
| $t$ | Time | [s] |
| $T$ | Temperature | [K] |
| $\mathbf{T}$ | Transformation matrix | [SI standard] |
| $u$ | Velocity | [m s$^{-1}$] |
| $u_x, u_y$ | $x$ and $y$ velocity components | [m s$^{-1}$] |
| $w_i$ | Weights | [−] |
| $\mathrm{e}^{❑}$ | Exponential function | [−] |
| $\mathrm{d}❑/\mathrm{d}t$, $\dot{❑}$ | Time derivative | [❑ s$^{-1}$] |

## Greek Letters

| Symbol | Definition | Unit |
|---|---|---|
| $\tau$ | Relaxation factor | [s] |
| $\nu$ | Kinematic viscosity | [m$^2$ s$^{-1}$] |
| $\varrho$ | Density | [kg m$^{-3}$] |
| $\mathbf{\Psi}$ | Collision matrix | [−] |
| $\omega$ | Collision frequency | [s$^{-1}$] |
| $\Omega$ | Collision operator | [SI standard] |

# Abbreviations

| Abbreviation | Definition |
|---|---|
| BC | Boundary Condition |
| BGKW | Collision model (Bhatnagar, Gross, Krook and Welander) [7, 8] |
| BUPC | Berkeley Unified Parallel C |
| CFD | Computational Fluid Dynamics |
| CGNS | CFD General Notation System |
| CPU | Central Processing Unit |
| FFT | Fast Fourier Transform |
| FLOPS | Floating Point Operations Per Second |
| GCI | Grid Convergence Index |
| GPGPU | General Purpose Graphical Processing Unit |
| HPC | High Performance Computing |
| LBM | Lattice Boltzmann Method |
| MPI | Message Passing Interface |
| MPPs | Massively Parallel Processors |
| MRT | Multiple Relaxation Time |
| NSE | Navier Stokes Equation |
| PGAS | Partitioned Global Address Space |
| RAM | Random Access Memory |
| TRT | Two Relaxation Time |
| UPC | Unified Parallel C |

# 1 Introduction

During the introduction, we will overview the objectives of this thesis and the history of parallel computing. The lattice Boltzmann method will be introduced to the reader.

## 1.1 Objectives of the Thesis

The main aim of this master thesis is to combine the strengths and parallelise the existing in-house lattice Boltzmann flow solvers [2, 9]. The method is well known for its high scalability [10]; therefore taking efforts to make the existing codes parallel is an important task. Such solvers are widely used in several fields, for example in automotive industry [11] or multiphase flows [12]. The formerly developed two-dimensional solvers [2, 9], which were written in C++ language, needs to be *reformulated* in C language to make the Partitioned Global Address Space (PGAS) [3, 13] method applicable. The benefit of using PGAS programming is that the code can run on multi-core and many-core architectures without the need of further code development or dedicated architecture code optimisation. The strengths of the codes will be *combined* and a new code will be implemented. The next step is to *parallelise* the solver, so that the calculation can be made using several central processing units. Until the end of the formerly described tasks; at several points *validation* might be required, since the code needs to be modified. From the previous studies, experimental and numerical data is available against which the new code can be validated.

The solver can be further improved by extending the possible application fields. One may apply it to resolve flows with different boundary conditions, multiphase flows, open surface flows, three-dimensional flows or for example different collision models might be required [12] in order to resolve properly a different flow with such solver.

The final outcome of the project is to *parallelise* the existing in-house codes such that the resulting code runs in parallel on CPUs, and resolves the same results as the serial codes do. In order to widen the potential of the code other physical phenomenons may be implemented and validated against numerical and experimental data as well.

## 1.2 Historical Overview of Parallel Computing

Parallel computing is a part in Computer Science discipline, where concurrent execution is studied. Both architecture and software developments are necessary to enable the efficient execution of a parallel program. During the last few decades, many researchers have worked on developing efficient software that can enable harvesting the performance offered by the new and more advanced hardware infrastructure. For better

understanding the benefits, aims and the need for parallel computing it is important to overview its history.

The basics of computers, which are used nowadays, is dated back to the 1940's, when John von Neumann, who was a Hungarian and American mathematician, physicist and inventor, laid down the main principles of operation for computers [14]. These were (a) the use of serial program execution, (b) using binary number system, (c) use of internal memory (RAM), (d) fully electronic operation and (e) the use of central processing unit (CPU).

The need for parallel computing was already emerged in the 1950's with advancements in the form of supercomputers [1]. These were used until the 1970's. Such computers were using *shared memory* multiprocessors and they were working side-by-side on shared memory.

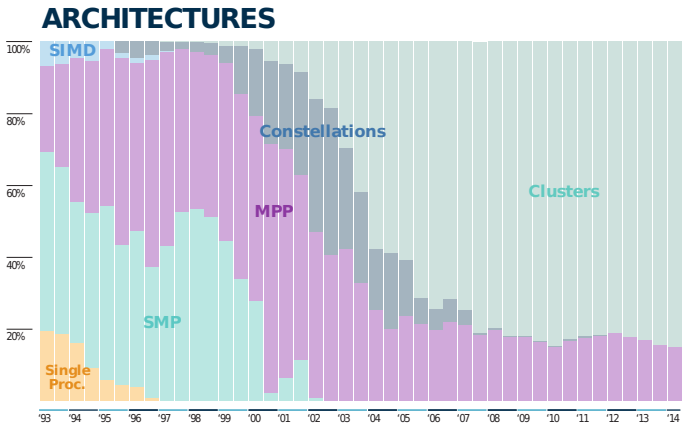



**Figure 1.1.** The TOP500 machines in terms of architecture from 1993 [1]

During the 80's, in the framework of the Caltech Concurrent Computation project, a supercomputer was built using 64 Intel processors. This approach, where massively parallel processors (MPPs) were used, determined the following years (see Figure 1.1). It can be seen in the figure that such systems were dominating in scientific computing until the 21$^{st}$ century. It is worth to note that the first one-trillion (one tera) floating point operations per second (TFLOPS) was achieved in 1997 by Intel (see Figure 1.2), while the first computers built in the 1950's were capable for 5 thousand FLOPS (kFLOPS) only [1]. This well represents the rapid increase in computing capability.

In the early 2000's, *clusters* replaced the MPP architectures. A cluster is a parallel computer built from a large number of computers. These are connected by a network. Each part is important in terms of clusters: the computers out of which such systems are built further evolved later on, and the interconnection between the so-called nodes (connected by the network) was also extensively developed later. Such architectures are used nowadays to perform high performance computing (HPC) tasks. The fastest

(highest number in terms of FLOPS) clusters are collected and listed in the so-called Top500 list [1], twice every year. This list is quickly changing. It is well known that the fastest machine is often replaced by a different machine by almost every report. The performance of the fastest computers in the recent years can be seen in Figure 1.2.

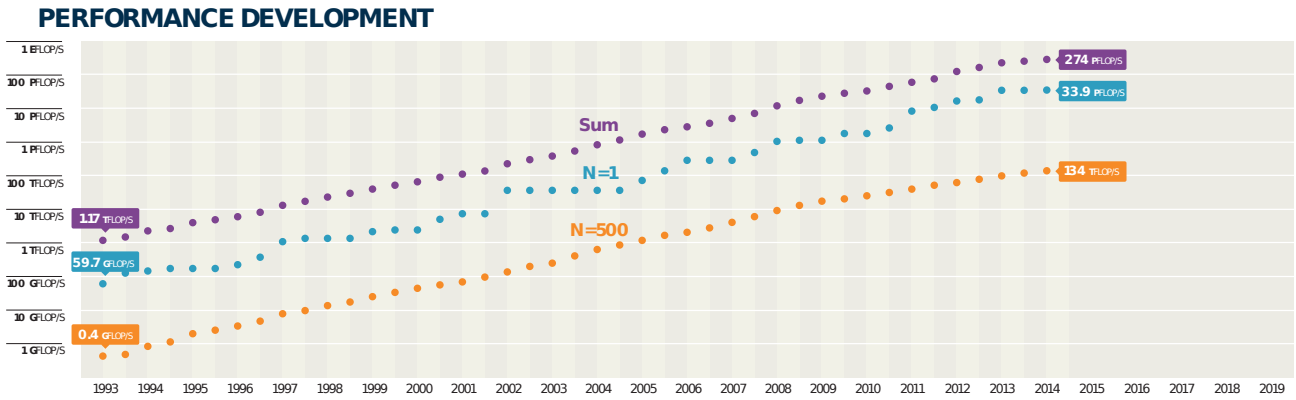



**Figure 1.2.** The TOP500 (best, 500th and summary) of machines as time evolved [1]

One wish to note that the Moore's law seemed to fail in the beginning of the 21st century. Moore's Law [15] is an empirical observation stated by Gordon E. Moore in 1965, who was one of the co-founder of Intel Corporation. The law states that the semiconductor industry is to *double* the performance of processors (i.e. number of transistors in a dense integrated circuit) in every 18 months [15]. In the early 21st century, this law seemed to be conserved only if the vendors increase the number of computing units (cores) in a certain CPU.

Nowadays, clusters involve a high number of computers connected, i.e. a large number of CPUs connected to each other. Within each CPU, one finds multiple cores, which are responsible for carrying out floating point operations. This well represents the great need for the efficient programming of such architectures. Without this, their potential can not be used.

The massive increase in the number of CPUs inherited the need for parallel software. These programs are much harder to write than sequential ones. This means that a certain task have to be divided between the working nodes and threads, which then need to communicate with each other. At certain points, during the calculation, synchronisation is usually necessary between the CPUs.

In the field of parallel programming, the Message Passing Interface (MPI) [16] was deterministic after the 1990's. In the mid to late 90's, the pthread and OpenMP [17] languages also showed up. These models were competing with each other in the recent decades. It is important to note that all of these (and many other but not necessarily widely used, more problem specific) models were based on the conventional serial programming languages, like for example C.

Nowadays, these models seem to be outdated and new approaches show up. One of these for example the architecture and programming approach developed by nVidia, which name is Cuda [18]. The basic idea of the vendor is to make the graphical programming units available not only for graphical purposes, but also for general programming purposes (General Purpose Graphical Programming Unit, GPGPU). In the framework of this thesis the so called Partitioned Global Address Space (PGAS) [19] approach is applied. This is also a novel technique in terms of parallel programming. The features of the applied PGAS language is discussed in Section 2.

One can see that there will not likely be a uniform solution for the parallel programming question. Just like one meets many different programming languages applied on many different devices (Java, C, Fortran, Python, C#, C++ ...). This is also a reason why one wish to deal and get familiar with the different approaches.

## 1.3 Overview of the Lattice Boltzmann Method

The lattice Boltzmann method became popular in the recent years in the community of Computational Fluid Dynamics (CFD) [20]. To highlight the reasons for this, one wish to understand the method itself and overview the advantages, disadvantages and the limitations of the approach.

One of the basic and well-known governing equation in fluid dynamics is the Navier-Stokes equation [21, 20]. The incompressible form of which is presented in Equation (1.1) [20].

$$\underbrace{\frac{\partial \underline{u}}{\partial t}}_{\text{I.}} + \underbrace{\nabla \cdot (\underline{u} \otimes \underline{u})}_{\text{II.}} = \underbrace{\underline{g}}_{\text{III.}} \underbrace{-\frac{1}{\varrho}\nabla p}_{\text{IV.}} + \underbrace{\nu \left(\nabla^2 \underline{u}\right)}_{\text{V.}} . \tag{1.1}$$

The equation has five terms. It describes 3D flow field of an incompressible (constant density $\varrho = \varrho_0$) fluid [20]. The first term is the local acceleration term, which presents only in unsteady flows. The second term is the convective term, which is a *non-linear* term and the most difficult to resolve during CFD simulations. The third term is the effect of the gravity (or acceleration), the fourth term considers the effect of the pressure gradient ($\nabla p$). The last (fifth) term is the diffusive term, which is originated to the viscosity ($\nu$) of the fluid. The Navier-Stokes approach considers the fluid as a *continuum*: it is usually applied on a control volume in the CFD practice [21].

Another representation of the motion of the fluid is to treat it at mesoscopic scales (in Greek "mesos" means middle) [12]. This approach is applied by the lattice Boltzmann method. The key of the approach is to represent the $x, y$ and $z$ velocity directions of the particles with the help of probabilities (distribution function). The governing equation can be derived using Figure 1.3. In the figure, one finds a particle, on which

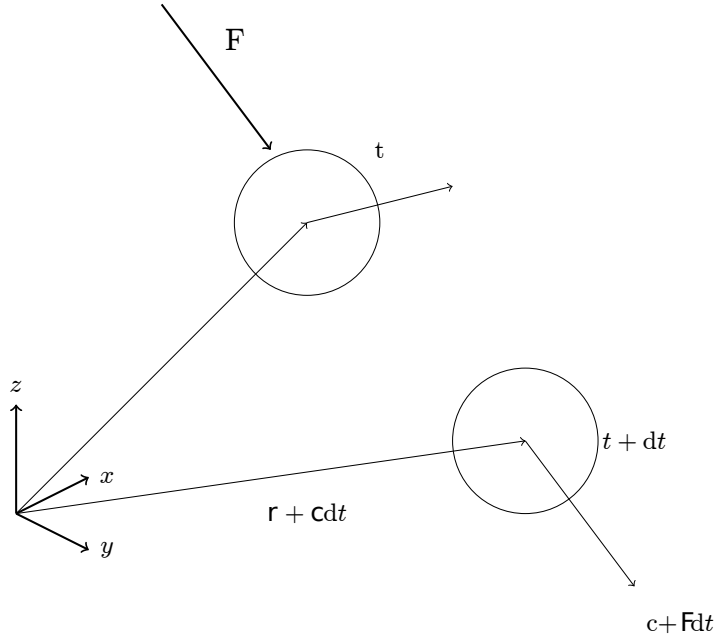



**Figure 1.3.** Motion of a particle [2]

an external force ($\mathbf{F}$) is applied. The mass of the particle is denoted by $m$. Based on the figure, one can write the following equation for the distribution function of the particles ($f$) [12]:

$$f(\mathbf{r} + \mathbf{c}\mathrm{d}t,\ \mathbf{c} + \mathbf{F}\mathrm{d}t,\ t + \mathrm{d}t) - f(\mathbf{r}, \mathbf{c}, t) = 0. \tag{1.2}$$

It is also important to note that the equation assumes that *no collision* takes place between the particles. The subtrahend ($f(\mathbf{r}, \mathbf{c}, t)$) shows the initial state described at a given position $\mathbf{r}$, where the particles move with a certain speed $\mathbf{c}$ at a certain time $t$. This state is changed in time by the external force $\mathbf{F}$. To take into account the effect of the collision, a collision operator needs to be introduced on the right hand side of the equation as a source term. Let us denote this by $\Omega$. The operator is responsible to change the velocity distribution function during time and space, i.e. $\mathrm{d}\mathbf{c}\mathrm{d}\mathbf{r}$. This means that the equation becomes [12]:

$$f(\mathbf{r} + \mathbf{c}\mathrm{d}t,\ \mathbf{c} + \mathbf{F}\mathrm{d}t,\ t + \mathrm{d}t)\mathrm{d}\mathbf{r}\mathrm{d}\mathbf{c} - f(\mathbf{r}, \mathbf{c}, t)\mathrm{d}\mathbf{r}\mathrm{d}\mathbf{c} = \Omega(f)\mathrm{d}\mathbf{r}\mathrm{d}\mathbf{c}\mathrm{d}t. \tag{1.3}$$

From this point, dividing by $\mathrm{d}\mathbf{r}\mathrm{d}\mathbf{c}$ and tending the time to zero the equation becomes

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \Omega(f). \tag{1.4}$$

Applying the chain rule for the left hand side, because it is still a function of time and

space yields

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial \mathbf{r}}\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} + \frac{\partial f}{\partial \mathbf{c}}\frac{\mathrm{d}\mathbf{c}}{\mathrm{d}t} + \frac{\partial f}{\partial t}\frac{\mathrm{d}t}{\mathrm{d}t} = \frac{\partial f}{\partial \mathbf{r}}\mathbf{c} + \frac{\partial f}{\partial \mathbf{c}}\mathbf{a} + \frac{\partial f}{\partial \mathrm{t}}, \tag{1.5}$$

where $\mathbf{a}$ is the acceleration defined by Newton's law as $\mathbf{a} = \mathbf{F}/m$ (constant mass). After replacing the left hand side one ends up with the following equation:

$$\frac{\partial f}{\partial \mathbf{r}}\mathbf{c} + \frac{\partial f}{\partial \mathbf{c}}\frac{\mathbf{F}}{m} + \frac{\partial f}{\partial \mathrm{t}} = \Omega(f). \tag{1.6}$$

The lattice Boltzmann approach assumes that no external force acts on the particles; therefore the equation written in its general form simplifies to

$$\boxed{\frac{\partial f}{\partial t} + \mathbf{c} \cdot \nabla f = \Omega(f).} \tag{1.7}$$

Note that Equation 1.7 is a scalar equation describing the velocity distribution function of particles. Compared to the Navier-Stokes equation, this is only a *scalar* equation and the non-linear *convective term vanished*. These properties makes the lattice Boltzmann method (LBM) computationally cheaper compared to the continuum approach. On the other side, the collision term, which is a function of the distribution function, is difficult to obtain. This is a drawback of the method. It is also important to note that based on the formerly noted properties, the LBM is easier to parallelise.

# 2 Literature Review

This section overviews the parallelisation models. It also introduces the possibilities given by the former in-house codes. The review shows the difference between the conventional and the novel parallelisation approach, which is applied in the framework of this thesis. We will see how others applied and what results they achieved using the new parallelisation model.

## 2.1 Formerly Developed Lattice Boltzmann Solvers

The properties of the lattice Boltzmann method has already been discussed in Section 1. The method itself has already been studied by several scientists [7, 8, 10, 12, 22]. This thesis relies on two, formerly developed in-house codes, which were developed by two former MSc students. Both Abbruzzese [9] and Teschner [2] developed their own two-dimensional lattice Boltzmann solver in 2013. Their codes form a basis for parallelisation, which is done in the framework of this thesis. During the thesis work, emphasis is put on the unification of the two codes considering the strengths of each solver.

Regarding the lattice Boltzmann approach, Teschner [2] stated that it is a valid alternative to the Navier Stokes equations (NSE). He also highlighted that it is a simpler representation describing the same complex physics problem (compared to NSE). The advantage of the method relies on that it uses a single equation to describe the flow. In this approach, the non-linear term vanishes. This term causes most of the difficulties in terms of solving the Navier Stokes equations. It means that the model is based on a linear partial differential equation, for which the solution is much easier to achieve. This property means that significantly less computational effort is required for such solver. The other advantage of the codes is their high scalability, i.e. certain parts of the code is highly parallelisable. Further advantage of the formerly developed codes is that their time step is always unity, i.e. the CFL number is the available biggest (with the applied explicit approach). This means that the problems, which were covered in the former theses, can be solved with the solvers in a robust and quick way. The only comparable approach with these solvers would be the implicit solution of the Navier Stokes equations for the investigated two-dimensional problems.

Both thesis noted that the codes suffer with higher number of required iterations beside using finer meshes. Tehschner [2] found that the required computational time with mesh size is exponential. One can assume that parallelisation can help winning over this issue. Both solver offers the same physical modelling approaches, i.e. two dimensional flows can be resolved with $D2Q9$ speed model (see later) and the same boundary conditions (BCs) were implemented to both of the solvers: inflow, outflow

(Zou/He type BC) and wall treatment (Bounce back BC, see in Section 3). The codes in terms of collision approximation also offer the same opportunities: Single Relaxation Time (BGKW) [7, 8], Multiple Relaxation Time (MRT) and Ginzburg [22] collision approximation (TRT). The mentioned models are discussed in Section 3. The results of both solvers were compared to available experimental results. It is also notable that handling curved boundary conditions gave challenges to the solvers, but the physical result earned for such cases were also acceptable; however the biggest errors were found in the regions of the curved walls.

## 2.2 Parallelisation Techniques

In the last decade, computers tend to have more processing units in a single CPU. This means that parallelisation of any program running on any type of computers became essential. In parallel with this, the computational power of clusters was enforced by the high number of processors applied in such systems. From the software side, this implied the treatment of multiple CPUs in the code. The two basic approach of parallel programming is the *shared memory* and the *message passing* approach. In the last decade, the *Partitioned Global Address Space* model showed promising results, which is a combination of the former two.

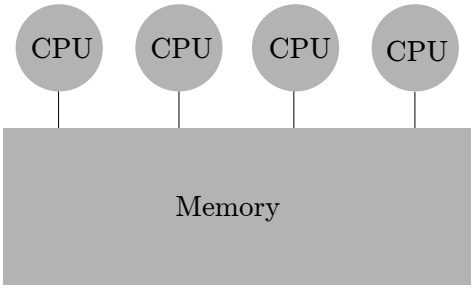



**Figure 2.1.** Shared memory model

The *shared memory* model [17] allows each working thread to access all the available data, i.e. the threads share the variables stored in the memory (see Figure 2.1). This approach is applied for example by OpenMP (Open Multi-Processing) [17]. The model is easy-to-program; however it allows only one-sided communication and it has no notion of data locality, i.e. remote access to the shared variables in execution time introduces reasonable overhead. This performance loss is caused because the programmers have no opportunity to describe how the data to be laid out in the address space.

The *message passing* model (Message Passing Interface: MPI, [16]) separates the memory to each working thread, i.e. a thread does not have access to the data created or stored by the other thread. All the data is local, therefore this approach exploits the

data locality. This offers fast memory access. The drawback of this approach is that it is more difficult to code, because in this model, whenever data needs to be exchanged between the nodes, communication is required. This needs to be implemented by the programmer. This shall be done by great care. The communication between the threads is two-sided, while in the shared memory approach the communication is one sided only. The main overhead in these codes is dedicated to the latency caused by data handling: transferring small amount of information between the working nodes (generally only limited information to be shared between the nodes). This causes the main performance loss of such code. Nota bene: This method have been extensively used in the past and it became a de facto standard in terms of parallel computing. The approach became highly mature in the last decades, or with other words it reached its limitations in terms of gaining further performance. In the last few years, other approaches showed up, which were dedicated to replace MPI.

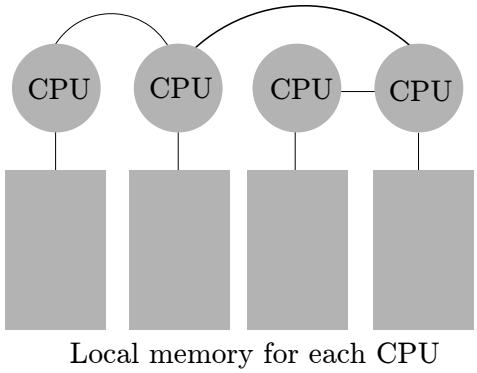



**Figure 2.2.** The message passing model

The *Partitioned Global Address Space* (PGAS) [19] model creates a bridge between the shared memory and message passing model. In this case, the programmer has the opportunity to define shared variables, which can be accessed by all the working threads. It is also possible to make advantage of the data locality by defining local variables reached by the "owner" node only. This means that passing messages between the nodes is also possible via the shared memory. As an extension to the C programming language, Unified Parallel C (UPC) was developed in the last decade. Other PGAS languages are for example Co-array Fortran, Titanium, X10 and Chapel. This thesis focuses on the UPC language.

The productivity of the UPC language was investigated by Cantonnet *et al.* [23] and they found that this model is easier to implement than the MPI approach. The drawback of the model is its complex memory representation, which requires data handling and at some point it may causes overhead.

The memory model of UPC can be seen in Figure 2.3, where each thread has an

associated private memory. Every thread is numbered, this number is a special constant in the programming language called `MYTHREAD`. The number of threads can be accessed by the variable `THREADS`. Two types of variable declaration is possible in the code, the first one takes place in the preprocessor stage, where global variables are declared as follows [13]

```
shared [block_size] type MyGlobalVar [number_of_elements];.
```

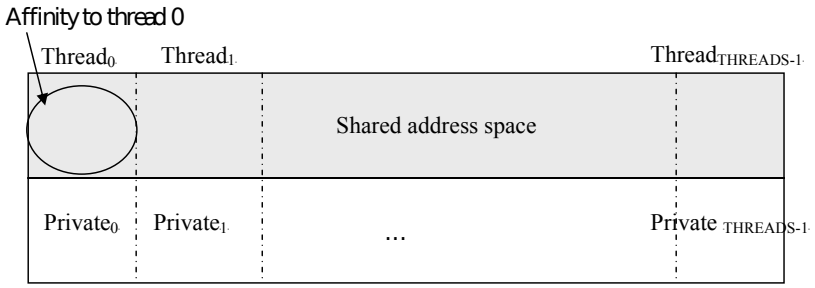



**Figure 2.3.** The memory model of Unified Parallel C (UPC) [3]

In this case, the vector called `MyGlobalVar` is located in the shared address space, and its elements are distributed among the threads with a round robin fashion, i.e. each `block_size` number of elements has an affinity to a certain thread, see Figure 2.4. Nota bene: The `block_size` in Unified Parallel C *must* be a compile time constant, i.e. before running the code and before compiling the code, the programmer must define a certain value for the block size. The figure shows the variable `array2` declared as follows: `shared [3] int array2 [10]`. The array has a length of 10 and the block size is 3. The default block size is one, i.e. the lack of the block size (or having `[]`) in declaration comes with block size of one [13].

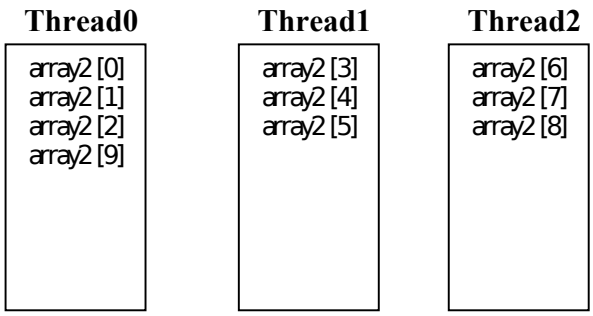



**Figure 2.4.** Affinity of global variable to threads: variable `array2` with a length of 10 elements declared with block size of 3 [3]

The local variables are declared within the `main` or other function according to the C coding standard [24], i.e.

```
type MyLocalVar [number_of_elements];.
```

The basic work-sharing function of UPC is the `upc_forall`, which is a special extension of the standard C `for` [24] cycle. `upc_forall` [3] has four fields:

```
upc_forall(int i=0; i<N; i++; &var[i]){ \* looped work here *\ },
```

i.e. the fourth field is an additional field to the standard `for` cycle. It is called affinity field [3, 13], which tells the compiler that which thread is dedicated to carry out the `i`$^{\text{th}}$ calculation. In this case, the thread which has an affinity to the `i`$^{\text{th}}$ element of `var`. The affinity field also accepts integer, i.e. one could use the loop variable `i` in this field. In this case the modulo division `(i%THREADS)` defines which thread shall carry out the calculations. It is important to note that the iterations of the loop must be independent of one another.

Up to this point, the discussed properties of UPC gives an opportunity to create the following loop, in which a simple vector addition is completed:

```
shared [n/THREADS] double vec1[n], vec2[n], vec3[n];
// ...
upc_forall(i=0; i<n; i++; &vec[i]){
    vec3[i] = vec1[i] + vec2[i];}
```

Pointer declaration in UPC is also possible, which is very similar to the standard C way [24]. The programmer has two types of memory and note that the pointer is located at a certain point, from where it points to another place. This means *four* types of pointers, which are declared as follows:

```
double *p2p;                     // private pointer to private memory
shared [bls] double *p2s;        // private pointer to shared  memory
double *shared s2p;              // shared  pointer to private memory
shared [bls] double *shared s2s; // shared  pointer to shared  memory
```

The first pointer is a typical C pointer `p2p`. The second pointer `p2s` is a local pointer pointing to the shared memory; therefore, with such declaration, each thread owns a copy of this pointer. The third pointer called `s2p`, is a pointer stored in shared space and it is pointing to the private memory. This should be avoided: the thread can reach only its dedicated (own) private memory space. The last pointer is a shared pointer (having block size of `bls`) pointing to the shared memory. Note that the block size is a compile time constant in terms of shared pointers (any pointer pointing to shared space) as well.

The private pointers are straight forward in terms of usage, but in UPC the pointers pointing to shared object are more sophisticated. The shared pointers has three fields: (a) thread affinity, (b) phase in the block and (c) the virtual address. The first one indicates that the pointed object has the affinity to which number of thread. The

second one indicates the position of the datum in the block (see Figure 2.5). Finally, the last field stores the address of the element. UPC allows to cast the type of pointers. This means that when we cast a shared pointer to a local one, the result will loose the thread information. El-Ghazawi *et al.* [25] state that casting a private pointer to a local one should be avoided. Such operation produces unknown results.

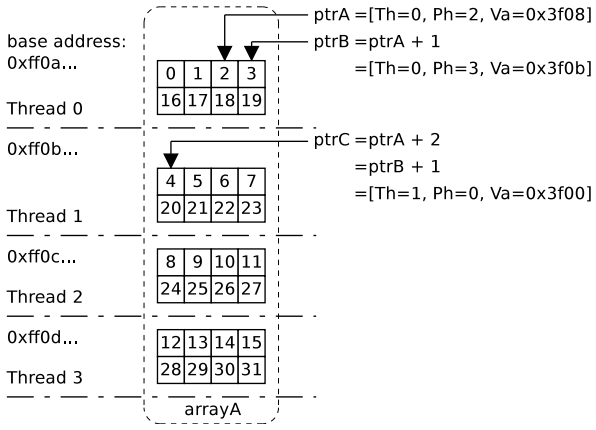



**Figure 2.5.** Shared pointers and their incrementation in Unified Parallel C [4]

Several papers showed that [25, 4, 26] the main overhead time is dedicated to the structure of shared pointers in UPC. Serres *et al.* [4] proposes a hardware support to decrease overhead. It is also noted that automatized optimization of the code (by the compiler) is not always helpful [4]. A more efficient approach is the optimization of the code done by the programmer, which requires the good understanding of the opportunities given by the UPC language. Serres *et al.* [4] discuss the drawback of the shared pointers (we mean pointers from local space to shared space), which is hidden in its incrementation. Whenever such pointer needs to be incremented, each of its field is incremented separately. In the this process, it is necessary to investigate the size of the blocks and the number of threads as well. The algorithm is complex involving all basic mathematical operations.

The incrementation of a pointer can be seen in Figure 2.5, where incrementing `ptrA` yields `ptrB` and incrementing `ptrB` yields `ptrC`. It can be seen that the pointer has three fields, and these fields are not necessarily change in every incrementation. The figure also gives a better understanding of phase property of a shared pointer, whereas the block size of the variable `arrayA` was four.

The variables in the local memory space can be allocated by the known C syntax (`malloc`) [24]. UPC requires the shared pointers to be allocated using the UPC syntax [13, 3]. The programmer has three ways to dynamically allocate memory in UPC. These are the follows: (a) `upc_global_alloc`, (b) `upc_all_alloc` and (c) `upc_alloc` [3]. All the pointers discussed in this paragraph are shown in Figure 2.6. For the exact

syntax used by these commands please refer to the UPC Manual [3]. One wish to note that `upc_global_alloc` will return allocated memory to the calling threads, i.e. this allocation is *not* a collective call. It means that when it is executed along all the threads, each thread is going to allocate the requested chunk of memory separately. Calling `upc_all_alloc` will result in one pointer available globally for all threads, i.e. it is a global call. This type of pointer becomes useful when collective operation is carried out on a variable. The last allocation offers an opportunity to allocate memory to each thread. This allocation is not a collective call, which returns a pointer pointing to the local shared memory, i.e. within the affined block. The allocated memory can be freed by executing the command `upc_free`, which is not a collective routine. This implies that for memory allocated by `upc_all_alloc` the freeing process is successful only if all the threads carry out the command.

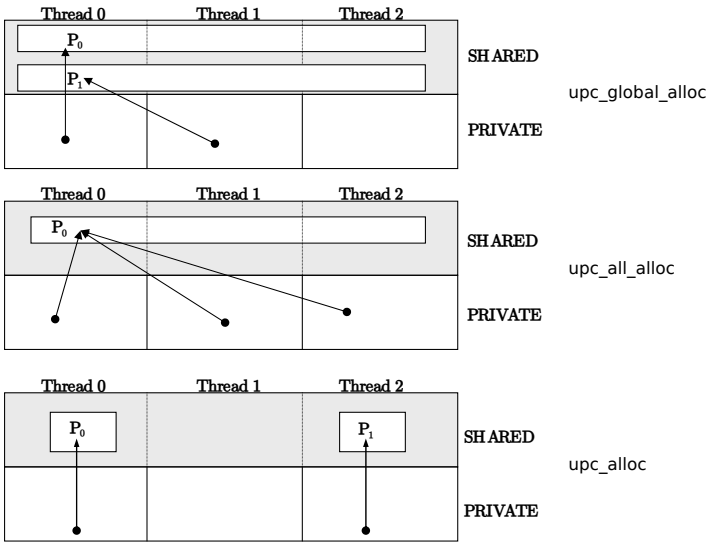



**Figure 2.6.** Types of different memory allocation in UPC [3]

Unified parallel C allows each thread to allocate private memory. This means that it also allows two-way communication between the working units. In terms of destination and source one can distinguish three types of copy: shared to shared, shared to local and local to shared. The commands executing these operations are respectively `upc_memcpy`, `upc_memget` and `upc_memput`. The usage is as follows [3]:

```
upc_memcpy(Dest, Source, NumOfElems) // copies from shared  to shared
upc_memput(Dest, Source, NumOfElems) // copies from private to shared
upc_memget(Dest, Source, NumOfElems) // copies from shared  to private
```

We can see that the three commands require the same arguments: a destination (where to copy), a source (from where to copy) and the number of elements (how many items to copy). Let us compare this to the MPI way, which reads as [16]:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int
    dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm, MPI_Status *status)
```

At first sight, we can see that *MPI is much more complicated*. In this case, a processor sends data to another processor. We need to clarify the data (what and how many elements to send), the type of the sent data, to which processor to send (`int dest`), with which tag to send it, through (usually) the MPI communication standard. This data needs to be received on the other side (receiving CPU), which needs to know that from which CPU is it receiving the data. It also returns a value (`status`), which indicates the success (or failure) of the receiving. We can conclude that the UPC approach is much simpler although it also needs two commands (put and get).

Mallon *et al.* (2009) [26] investigated the performance of UPC in terms of communication and compared the results with MPI. It was found that the overhead time is a function of the memory architecture and the message size, and the communication pattern of primitives. UPC showed poor results in collective performance compared to MPI [27], due to high start-up communication latencies. They also concluded that UPC can take a full advantage of efficient and scalable primitives and collectives. In other cases, their UPC code showed a better performance than MPI.

UPC offers two types of memory consistency. The first one is the strict and the second one is the relaxed. The default is the relaxed, where the compiler applies no restriction in terms of access and overwriting the shared variables. Strict consistency can be applied along the whole code, in a block or in an object (`#define` and `#pragma`). In strict case sequential execution is enforced on shared data.

```
#include <upc.h>         // relaxed consistency: full program level
#include <upc_relaxed.h> // relaxed consistency: full program level
#include <upc_strict.h>  // strict  consistency: full program level
```

Thread synchronization is also possible in UPC, which can be achieved via locks and barriers. This offers the programmer the opportunity to execute sections of the code in a mutually exclusive fashion. The concerning commands are the follows: `upc_lock`, `upc_unlock` and `upc_lock_attempt` [3]. The UPC language also provides blocking and non-blocking barriers. The basic barrier is `upc_barrier` which is dedicated to synchronize the threads at runtime. Note that the command `upc_fence` ensures that until a this point (position of the fence), all the former shared variable based operations are performed.

The productivity of UPC language was investigated by Cantonnet *et al.* (2004) [23]. Namely, they investigated the manual programming effort. They showed the required number of lines and required number of characters to implement different codes, which

are generally used for benchmarking. They stated that UPC showed consistent improvement over MPI in three points of view. They found that implementing a UPC code requires less lines, less characters and also less conceptual effort. One having experience with MPI may also agree with these statements after producing some basic codes with UPC.

Unified Parallel C was developed in a consortium, in which US government, industrial participants and academia took effort. This helped a lot to make UPC more and more widely used in high-performance computing. As a result, one finds several UPC compilers available, which are the follows:

- GNU UPC: `http://www.gccupc.org/`, linked with the open source compiler `gcc`,
- HP UPC compiler (commercial),
- IBM XL UPC compiler (commercial),
- Berkeley UPC compiler: `http://upc.lbl.gov/` can be linked with Intel C compiler, open source,
- Michigan Tech MuPC `http://www.upc.mtu.edu/` and
- Cray UPC compiler.

Cranfield University holds a licence for Intel C compiler, to which the Berkeley UPC (BUPC) compiler can be linked (BUPC first transforms UPC to C). Therefore, it is a reasonable choice to use the combination of the two compilers during the studies carried out in the framework of this thesis. The performance of the chosen Berkeley UPC compiler was evaluated by Husbands *et al.* (2003) [5]. The overall structure of Berkeley UPC compiler can be seen in Figure 2.7. The three main components of the system are: (a) UPC-to-C translator, (b) the runtime system and (c) the GASNet (Global Address Space Net) communication system. The system is modular and supports different C compilers, unlike the commercial compilers.

Husbands *et al.* [5] compare the performance of Berkeley UPC compiler to the HP UPC compiler. It was found that Berkeley UPC (BUPC) performed well on the synthetic benchmarks. It showed a good communication performance, which reason was hidden in the architecture of the compiler: GASNet [28] is responsible for the lowest level communication. The serial codes showed good performance also. Husbands *et al.* [5] stated that their compiler (BUPC) is competitive with the available commercial compilers in relative sense since it was dominating over the performance measurements. The paper states that the compiler achieves high performance in pointer-to-shared arithmetic because of their own compact pointer representation. They also found that

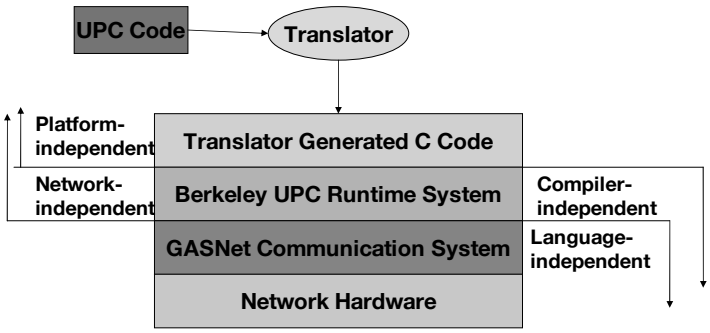



**Figure 2.7.** The architecture of Berkeley UPC compiler [5]

BUPC performed better in solving small message traffic based problems than the HP UPC compiler. The article was written in 2003, when the mentioned compilers were "maturing" [5]; therefore one can assume that in one decade time the investigated compilers are already essential tools for solving problems in parallel computing.

Most of the formerly discussed papers apply synthetic benchmarks (FFT calculations, N-Queens problem, NAS benchmarks from NASA etc.) which is not necessarily relevant for this thesis. A limited number of papers use the UPC as a tool for solving physics related problems. Markidis and Lapenta (2011) [29] used UPC for implementing a particle-in-cell code to simulate plasma. They experienced performance degradation for high number of CPUs. The effect was dedicated to a specific part of their solver. They clearly stated that UPC is an appropriate tool for solving such problem with parallel approach.

Johnson (2005) [30] used Bekeley UPC compiler on a Cray machine [31] to solve Computational Fluid Dynamics (CFD) problems. The used solver was an in-house code. Comparisons were made in terms of performance to MPI. Two flows were simulated, one around an unmanned aircraft and an other one around a military ground vehicle. The UPC based code showed better performance than the MPI version of the code. The difference in the performance was bigger for higher number of CPUs: UPC performed better than MPI, especially above 64 CPUs (the comparison was made between using 4 to 124 CPUs). The study measured the communication time as a function of message size. It was shown that MPI required more time to pass small size of messages than UPC. Above a certain message size, UPC still performed better, but the difference was negligible. As the study was presented in 2011, one may assume that UPC finally became a more or at least as powerful tool for parallel computing as MPI.

# 3 Methodology

In this section we will see the how the numerical lattice Boltzmann method works. The section introduces the collision and speed models, boundary conditions, the connection of the method with the macroscopic world and finally, the advantages and the limitations of the modelling.

## 3.1 Numerical Approach of the Lattice Boltzmann Method

We have seen the lattice Boltzmann approach in Section 1. One needs to discretize Equation (1.7) (repeated below) to achieve numerical results. The collision operator must be defined too. In the followings, the numerical approach of the LBM to be discussed. On the numerical side, the LBM consist of three main steps, which are (a) the collision step, (b) the streaming step and finally (c) the calculation of the macroscopic variables $(\mathbf{u}, \varrho, p)$.

$$\frac{\partial f}{\partial t} + \mathbf{c} \cdot \nabla f = \Omega(f).$$

The need of discretisation applies for the LBM too, similarly as in the continuum approach. However, the way of discretisation is different: the lattice Boltzmann method does *not* require the approximation of derivatives. Instead of calculating derivatives (as it is done during solving the Navier-Stokes equation), in this case *speed models* are used. These models are responsible to describe how the particles collide. Such models use the equilibrium distribution function $f^{eq}$, which was derived by Maxwell and Boltzmann [32, 33, 34]. This reads as followings:

$$f^{eq}(\mathbf{c}) = 4\pi \left( \frac{m}{2\pi k_B T} \right)^{\frac{3}{2}} \mathbf{c}^2 \mathrm{e}^{\frac{-m\mathbf{c}^2}{2k_B T}}, \tag{3.1}$$

which describes that along any direction ($x, y$ and $z$) what is the *probability* of finding a particle travelling with velocity $\mathbf{c}$. The expression depends on the molar mass ($m$), temperature ($T$) and velocity ($\mathbf{c}$). $k_B$ is the Boltzmann constant. Note that no spatial dependence present. Omitting the derivation (for the detailed derivation please refer to [2]), approximating the equilibrium distribution function with *second order truncation error* reads as

$$f^{eq} = \varphi w_i \left[ 1 + \frac{c_i \cdot \mathbf{u}}{c_s^2} + \frac{(c_i \cdot \mathbf{u})^2}{2c_s^4} - \frac{u^2}{2c_s^2} \right], \tag{3.2}$$

where $w_i$ are the corresponding weights for the used speed model (see later), $\varphi$ is the generalized variable for which the equilibrium distribution function is applied (e.g. velocity). $c_i$ is the velocity component in the $i^{\text{th}}$ direction and $c_s^2$ is the speed of sound in

so called lattice units (since the method is used in a dimensionless mesh called lattice). *Nota bene*: The order of accuracy of the method is given by this discretisation, since no derivatives are approximated to solve Equation 1.7. The speed of sound within the lattice is equal to $c_s = 1/\sqrt{3}$ [12].

### 3.1.1 Collision Models

The biggest step towards the numerical solution of Equation (1.7) is to discretise the collision operator $\Omega$. It is a function of the distribution function, i.e. $\Omega = \Omega(f)$. The first model used in the framework of this thesis is named BGKW collision model. It is actually named after scientists who derived the model independently from each other. The model was introduced in 1954 by Bhatnagar, Gross and Krook [7], and in the same year by Welander [8]. This model is classified as a single-relaxation time scheme. The expression for the collision operator in case of the BGKW model reads as

$$\Omega(f) = \omega(f^{eq} - f) = \frac{1}{\tau}(f^{eq} - f), \tag{3.3}$$

where $\omega$ is the collision frequency calculated as the reciprocal of the relaxation factor $\tau$. Finally, $f^{eq}$ is the local equilibrium distribution function, which was discussed formerly. The collision frequency can be calculated by knowing the kinematic viscosity of the fluid ($\nu$) according to Equation (3.4).

$$\omega = \frac{1}{3\nu + 1/2} \tag{3.4}$$

The next collision model to discuss is denoted as TRT, which is the abbreviation of two-relaxation time. As it is seen in Equation (3.4), as the viscosity decreases, the source term (collision operator) increases, which leads to numerical instabilities for the BGKW model [7, 8]. The two-relaxation time model is dedicated to reduce this instability, which was introduced by Ginzburg [22]. The basic idea was to resolve the distribution function with two parts: (a) a symmetric part, and (b) an asymmetric part. The definition of the two parts ($f^s$ & $f^a$) are [22]

$$f^{eq} = f^s + f^a = \underbrace{\frac{1}{2}\left(f + \hat{f}\right)}_{f^s} + \underbrace{\frac{1}{2}\left(f - \hat{f}\right)}_{f^a}, \tag{3.5}$$

where $\hat{f}$ is the distribution function which is travelling in the opposite direction with respect to $f$. The collision frequencies are calculated respectively, as [22]

$$\omega_s = \omega, \tag{3.6}$$

and

$$\omega_a = \frac{8(2-\omega_s)}{8-\omega_s}. \tag{3.7}$$

The last collision model discussed in the framework of this thesis is the so-called multi-relaxation model or MRT [2, 35, 36, 37]. This approach replaces the collision frequency with the collision matrix denoted by $\mathbf{\Psi}$, i.e. Equation (1.3) becomes [36]

$$f(\mathbf{r}+\mathbf{c}\mathrm{d}t,\ \mathbf{c}+\mathbf{F}\mathrm{d}t,\ t+\mathrm{d}t) - f(\mathbf{r},\mathbf{c},t) = \mathbf{\Psi}\left[f^{eq}(\mathbf{r},\mathbf{c},t) - f(\mathbf{r},\mathbf{c},t)\right]. \tag{3.8}$$

Replacing the matrix with $\omega$ yields the BGKW collision model. It is a must to conserve momentum, which can be described only using momentum space variables. This implies that the collision matrix needs to be transformed to momentum space. The transformation is done via a proper matrix multiplication, i.e. the collision matrix in momentum space reads as $\hat{\mathbf{\Psi}} = \mathbf{T}\mathbf{\Psi}\mathbf{T}^{-1} = \mathbf{T}^{-1}\mathbf{S}$, where $\mathbf{S}$ is the relaxation matrix, which is diagonal, while $\mathbf{\Psi}$ is a square matrix. Transforming $f$ to momentum space yields momentum vector $\mathbf{m_v} = \mathbf{T}f$. Using these relations and putting them back to Equation (3.8) yields

$$f(\mathbf{r}+\mathbf{c}\mathrm{d}t,\ \mathbf{c}+\mathbf{F}\mathrm{d}t,\ t+\mathrm{d}t) - f(\mathbf{r},\mathbf{c},t) = \mathbf{T}^{-1}\mathbf{S}\left[\mathbf{m_v^{eq}}(\mathbf{r},\mathbf{c},t) - \mathbf{m_v}(\mathbf{r},\mathbf{c},t)\right]. \tag{3.9}$$

In order to define the elements of the matrix, one needs to know the used speed model during the calculation. The speed model is discussed in the followings. It is important to note, that we can see, that out of the three discussed collision models, this is the *most complex* and *most computational expensive*, i.e. the slowest to run. At the same point one wish to achieve the most accurate results with this model. Further informations on the model can be found in [35, 36, 37].

### 3.1.2 Speed Models

The next step is to choose the dimensions of the flow to be described: 1, 2 or 3 dimensional, and choose which speed model to use. In the lattice Boltzmann method, this is denoted as *DaQb*, where *a* stands for the number of dimensions and *b* shows the speed model. Number *b* describes that in *how many directions does the distribution function f is discretised inside a lattice*. In the followings, with the help of a few figures, these speed models are discussed.

In one dimension, two basic possibilities are given [2], which is shown in Figure 3.1. It is important to note that from fluid dynamics point of view, it is required to have the length of the sites to be equal. In the framework of this thesis (and in other solutions for the same problem too) the time step (Courant-Friedrichs-Lewy condition [38]) is unity and the lattice (mesh) is dimensionless, which means that d$t$ =d$x$=1. The basic

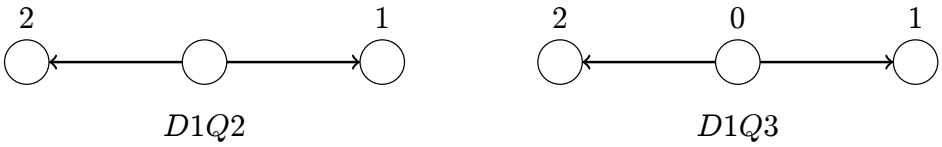



**Figure 3.1.** Two of the possible speed-models in one dimension [2]

| $D2Q9$ | | |
|---|---|---|
| direction | $w_i$ | $c_i$ |
| 0 | 4/9 | (0,0) |
| 1 | 1/9 | (0,1) |
| 2 | 1/36 | (1,1) |
| 3 | 1/9 | (1,0) |
| 4 | 1/36 | (1,-1) |
| 5 | 1/9 | (0,-1) |
| 6 | 1/36 | (-1,-1) |
| 7 | 1/9 | (-1,0) |
| 8 | 1/36 | (-1,1) |

**Table 3.1.** $2DQ9$ speed model details: $c_i$ (directions) and $w_i$ (weights) [2]

difference between the two shown 1D models is that in the $D1Q3$ case the fluid has the opportunity to stay at rest.

One has more opportunities in two dimensions. The basic and most widely used approach is the $D2Q9$ model [35, 2, 9, 10], which is applied in the framework of this thesis. A few two dimensional speed-models are shown in Figure 3.2. The second and the third shown speed-model allows the particle to stay at rest ($c_i = (0, 0)$). The properties of the speed models are listed in Table 3.1, where one finds the velocity components $c_i$ (directions) and the weights $w_i$.

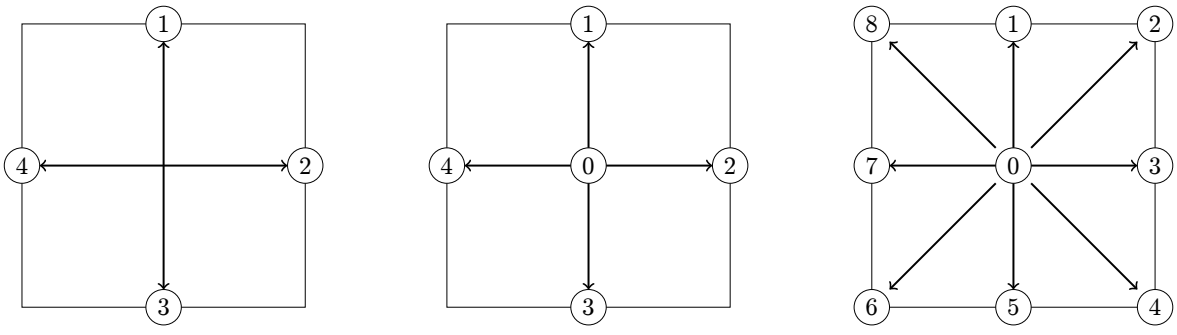
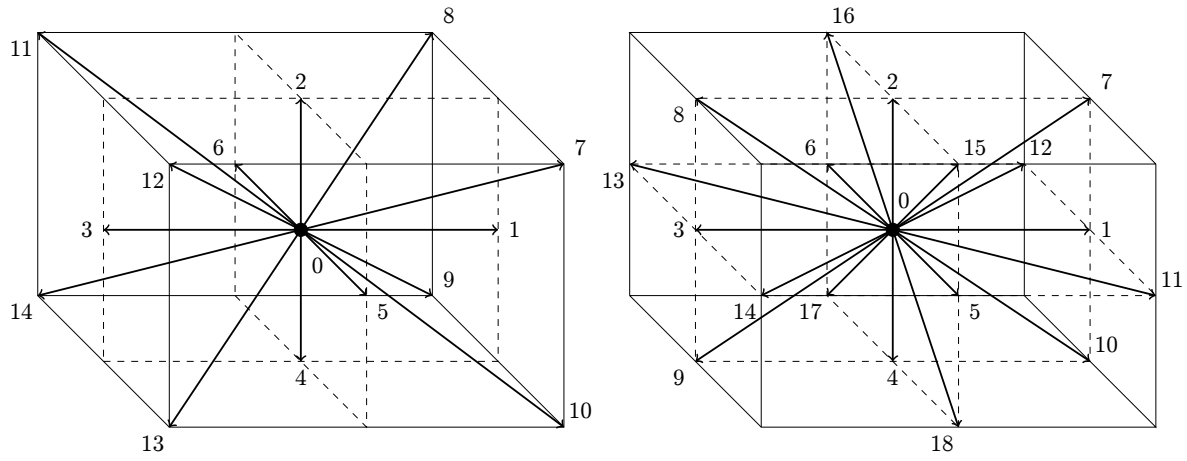



**Figure 3.2.** Three of the possible speed-models in two dimension [2]

Finally, the three dimension lattice arrangements (speed-models) are introduced

[10]. This is the most complex case. Two of the possible solutions are shown in Figure 3.3. One can see that the number of directions rapidly increase in 3D case, which also implies more computation at a single step.




*D3Q15* *D3Q19*

**Figure 3.3.** Two of the possible speed-models in three dimension [2]

### 3.1.3 Macroscopic Variables

It is an important question *how macroscopic variables are calculated* from the function [10]. In the followings, one can see the equations with which the macroscopic variables can be calculated. These are the followings: (a) density, (b) pressure and (c) $x$ and $y$ velocity components.

(a) The density is the sum of the resolved distribution function in all the directions [10], i.e.

$$\varrho(\mathbf{r}, t) = \sum f(\mathbf{r}, t) \tag{3.10}$$

(b) The pressure is the third of density [10], since $c_s = 1/\sqrt{3}$:

$$p(\mathbf{r}, t) = c_s^2 \varrho(\mathbf{r}, t) = \frac{\varrho(\mathbf{r}, t)}{3} \tag{3.11}$$

(c) The $x$ and $y$ velocity components can be calculated from the $x$ and $y$ scalar product of the directions ($c_i$) and the distribution function ($f_i$) [10], i.e.

$$\varrho(\mathbf{r},t)u_x(\mathbf{r},t) = \sum c_x(\mathbf{r},t)f(\mathbf{r},t) \text{ therefore} \tag{3.12}$$

$$u_x(\mathbf{r},t) = \frac{\sum c_x(\mathbf{r},t)f(\mathbf{r},t)}{\varrho(\mathbf{r},t)} \text{ and} \tag{3.13}$$

$$\varrho(\mathbf{r},t)u_y(\mathbf{r},t) = \sum c_y(\mathbf{r},t)f(\mathbf{r},t) \text{ from which} \tag{3.14}$$

$$u_y(\mathbf{r},t) = \frac{\sum c_y(\mathbf{r},t)f(\mathbf{r},t)}{\varrho(\mathbf{r},t)}. \tag{3.15}$$

Finally, it is important to note that the modelling approach is *dimensionless*. This means that the Reynolds number has a slightly different way of definition. Compared to the classical way how Reynolds number is calculated, in this case instead of the length one needs the *number of nodes*. Therefore, the ratio of inertial forces to viscous forces ($Re$) is defined as

$$Re = \frac{N_L u_L}{\nu_L}, \tag{3.16}$$

where $N_L$ is the number of nodes, e.g. in the lid-driven cavity validation case this is the number of nodes along the width (or height) of the domain (since the aspect ratio is unity). $u_L$ is the characteristic lattice velocity and $\nu_L$ is the lattice viscosity.

As we have seen formerly, the method is an unsteady approach. This can be seen by the presence of the first term in Equation 1.7, which is the unsteady term. This means that during the calculations a proper time-step is needed. In addition to the dimensionless modelling approach, the time step of the method is always unity in terms of mesoscopic (simulated) scales. This means, that after the collision process, the calculated velocities are streamed to the neighbouring cells. Therefore, the $\Delta t$ (time-step) is equal to the lattice size $\Delta x$ over unit velocity $c = 1$, i.e. $\Delta t = \Delta x/c = \Delta x$ [10].

### 3.1.4 Boundary Conditions

In terms of boundary conditions, the method requires some special treatment. This means that the restrictions given by the macroscopic quantities (e.g. given velocity) have to be "translated" to the "language of the lattice". In the followings, the *wall, inflow* and the *outflow* boundary conditions will be discussed.

In the framework of this thesis a two-dimensional solver is discussed using $D2Q9$ speed-model. Based on this, the boundary conditions will be introduced using examples which are based on this speed model. The first boundary condition, which is usually the basic in terms of CFD is the wall boundary condition [39]. In the lattice Boltzmann approach it is called *bounce back*. Figure 3.4 shows how the boundary is treated. Taking a wall which is south from a node, the solver has no information about the flow (nothing to stream) from southward. To ensure that the velocity of the wall is zero and no flow

occurs through the wall, the distribution function in the appropriate directions must be equal. In the example shown in the figure, this means that $f_1 \equiv f_5$, $f_2 \equiv f_6$ and $f_8 \equiv f_4$. With such approach one can define a no slip condition wall. Defining these constraints yields that the distribution function bounces back from the wall. Note that the conservation of momentum is automatically satisfied since no elastic work is considered in the constraint.

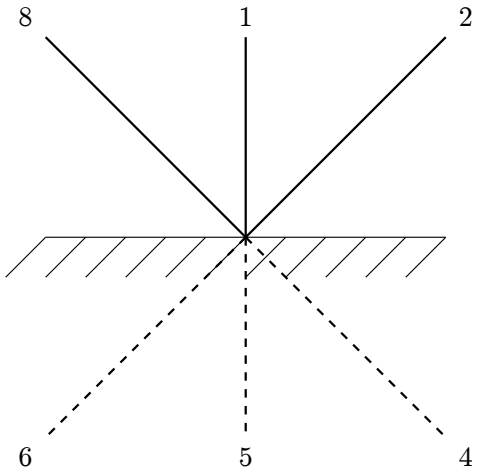



**Figure 3.4.** Bounce back (wall) boundary condition [2]

Zou and He further developed the bounce back boundary condition [40] The result of their work gave the possibility to define *inlet* or *outlet* boundary conditions. They defined these boundary conditions in a universal form, which means that it can be implemented for any collision model beside any speed-model. For this reason this approach is very often used in lattice Boltzmann solvers [2, 9, 39]. Let us consider an inlet condition as it is shown in Figure 3.5. In this case, we want the flow to enter the domain from the left hand side. There is no information available about the discrete directions of distribution function shown with dashed line ($f_2, f_3$ and $f_4$). The main idea of Zou and He [40] was to *bounce back the non-equilibrium part of the appropriate distribution functions*, i.e. based on the figure this means that [40]

$$f_2 - f_2^{eq} = f_8 - f_8^{eq}, \qquad f_3 - f_3^{eq} = f_7 - f_7^{eq} \text{ and} \qquad f_4 - f_4^{eq} = f_6 - f_6^{eq}.$$

Note that the macroscopic variables can be computed as

$$\sum f(\mathbf{r}, t) = \varrho(\mathbf{r}, t), \tag{3.17}$$

$$\sum c_x(\mathbf{r}, t) f(\mathbf{r}, t) = \varrho(\mathbf{r}, t) u_x(\mathbf{r}, t) \text{ and} \tag{3.18}$$

$$\sum c_y(\mathbf{r}, t) f(\mathbf{r}, t) = \varrho(\mathbf{r}, t) u_y(\mathbf{r}, t). \tag{3.19}$$

Zou and He assumed *constant density* ($\varrho = \varrho_0$) at the inlet. Based on these assumptions it is possible to define the unknown distribution functions ($f_2, f_3$ and $f_4$) via the equations of the macroscopic variables (Equations (3.10),(3.13),(3.15)). First, let us define the density with the help of Equations (3.10) and (3.13). This yields

$$\varrho = \frac{1}{1-u}\left[f_0 + f_1 + f_5 + 2(f_6 + f_7 + f_8)\right]. \tag{3.20}$$

With the help of Equation (3.2) one can obtain the following relations:

$$f_2 = f_6 - \frac{1}{2}(f_1 - f_5) + \frac{1}{6}\varrho u - \frac{1}{2}\varrho v, \tag{3.21}$$

$$f_3 = f_7 + \frac{2}{3}\varrho u \text{ and} \tag{3.22}$$

$$f_4 = f_8 - \frac{1}{2}(f_1 - f_5) + \frac{1}{6}\varrho u - \frac{1}{2}\varrho v. \tag{3.23}$$

Based on these expressions, one can define inlet conditions; furthermore with such approach arbitrary Dirichlet conditions can be derived (e.g. for pressure). There is also an option to create outlet condition based on this approach; however the assumption of constant density at the downstream end might not be acceptable.

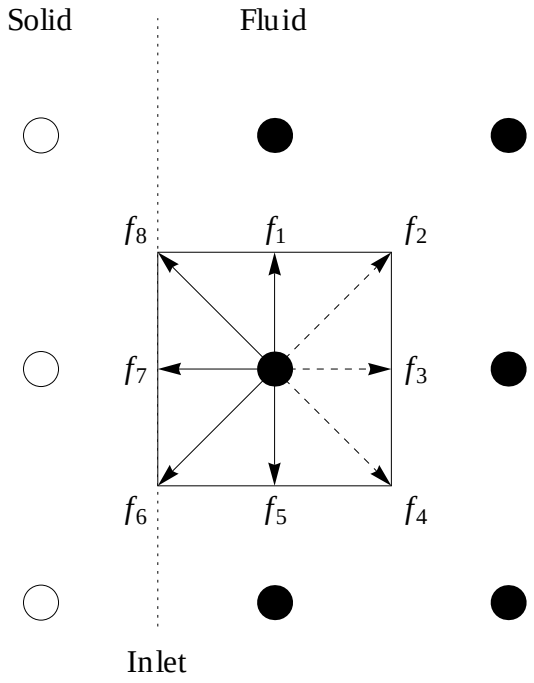



**Figure 3.5.** Inflow boundary condition from the left hand side

It was reported in the former theses [2, 9] and also by others [12] that the definition

of proper outflow condition is one of the most challenging tasks in the method. A more likely stable and appropriate definition of outlet, compared to the Zou and He approach, is available if one defines *opening boundary*. The basic idea is simpler compared to the formerly seen Dirichlet condition. In this case, the unknown distribution functions are *extrapolated* based on the upstream flow data. Based on Figure 3.6, one can define the second order approximation of the unknown distribution functions at the $n^{\text{th}}$ node (assuming $n$ nodes in $x$ direction). Note that the second order approximation is required to keep the second order accuracy of the method. The equilibrium distribution function was resolved with second order accuracy. The unknown directions can be calculated as

$$f_{8,n} = 2f_{8,n-1} - f_{8,n-2}, \tag{3.24}$$

$$f_{7,n} = 2f_{7,n-1} - f_{7,n-2} \text{ and} \tag{3.25}$$

$$f_{6,n} = 2f_{6,n-1} - f_{6,n-2}. \tag{3.26}$$

Muhamad reported [12] that the method can be unstable in some conditions, where the first order approximation can overcome the issue. For the first order approximation one can define equality between the $n^{\text{th}}$ and $(n-1)^{\text{th}}$ nodes.

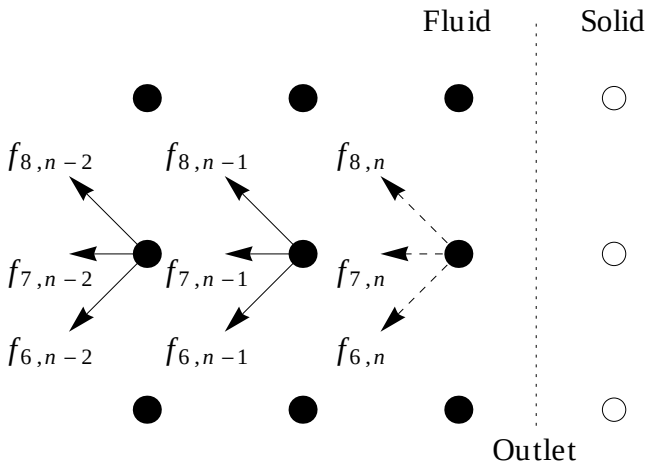



**Figure 3.6.** Outlet boundary condition

Finally, one wish to take some notes on the corner treatment. This topic is not discussed in the literature, but it can be a source of error if it is not treated properly [2, 9]. In corner points, the distribution function is available from two directions. The problem rises when two different boundary conditions meet at the corner: i.e. stationary wall & moving wall meets. In the case when east and north walls meet (see Figure 3.7), variable $f_6$ can be treated either as moving or stationary wall. The issue rises for

example in the lid driven cavity validation case (see later). In the lid driven cavity, the improper handling of such boundary can eliminate the corner vortices in case of small Reynolds number, which means non-physical results. In order to resolve the physics properly, it was found [2, 9] that the corner points shall be treated as walls or in other words zero velocities has to be prescribed. By this approach the stability of the solver and the physical results can be kept at the same time.

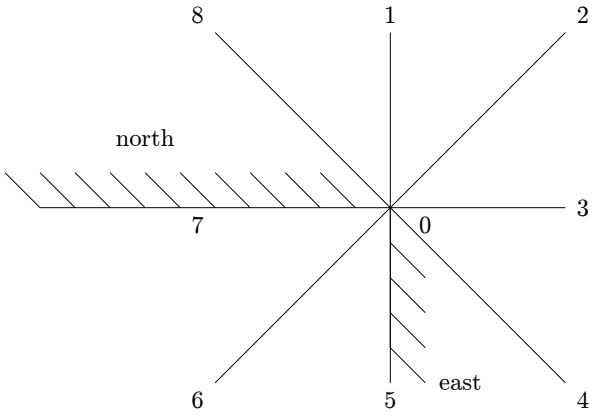



**Figure 3.7.** Boundary conditions in the corner points: east and north walls [2]

### 3.1.5 Residual Calculation

During the iterative process (at each time-step), it is important to monitor the "goodness" of the simulations. For this purpose, residuals can be calculated. The residuals were defined in the same way as in the former codes [2, 9]. In order to calculate the residuals, first the velocity magnitudes and densities were summed for all the cells. The summation was made at the $k^{\text{th}}$ iteration (time step). The residuals were calculated such that they show the *relative change* of sum of velocity and density respectively, see Equation 3.29 and 3.30.

$$S_{vel,k} = \sum_{i=1}^{n \cdot m} \sqrt{u_i^2 + v_i^2} \tag{3.27}$$

$$S_{\varrho,k} = \sum_{i=1}^{n \cdot m} \varrho_i \tag{3.28}$$

$$R_{vel,k} = \frac{S_{vel,k-1} - S_{vel,1}}{\max(S_{vel,k-1}, S_{vel,1})} \tag{3.29}$$

$$R_{\varrho,k} = \left| \frac{S_{\varrho,k-1} - S_{\varrho,1}}{\max(S_{\varrho,k-1}, S_{\varrho,1})} \right| \tag{3.30}$$

### 3.1.6 Advantages and Disadvantages

Finally, let us consider the advantages and the limitations of the numerical lattice Boltzmann approach. The first to note is that the meshing can be automatized, since the mesh is uniform. Compared to other meshing approaches, in this case the meshing can be parallelised too. The method is relatively easy to implement because of its features: explicit, (basically) two step method. The approach showed to be reasonable in terms of solving multiphase flows [41, 42], Rayleigh-Taylor instability [42], blood flow [43, 44, 45] simulations etc. From the limitation side, we have to note the relatively high memory requirements ($D2Q9$: store distribution function in 9 directions) and the limited applicability to incompressible flows.

# 4 Results and Discussion

In the following section, we will discuss how the available codes were used: the reformulation of the C code and its parallelisation. The new C code will be introduced to the reader (new features, advantages and disadvantages). Once the C code was ready, the parallelisation was the next step. For this purpose, a preliminary benchmark was carried out, which showed some properties of Unified Parallel C. In this section, we will see the way of parallelisation and the gained speed-ups. Finally, the validation of the new codes is presented.

## 4.1 Preliminary Parallel Benchmarks

As it was discussed formerly in Section 2, Unified Parallel C offers different ways to handle and manage data. The threads can access variables stored in shared space or in local space. The data can be a pointer (allocated dynamically) or a static variable. This means that mainly the following four type of variables can be defined:

- static variable in local space (`void MyLocalStaticVar`),
- pointer to local space (`void *MyLocalPointer`),
- static variable in shared space (`shared [] void MySharedStaticVar`) and
- pointer to shared space (`shared [] void *MySharedStaticVar`).

Two benchmark codes were created (see in Appendix A), which were supposed to increment the value of each element in a vector for certain times. All of the four data types were tested, and the profiling of the codes were carried out. Calculations were made with a serial code, which was compiled with Intel C compiler (`icc`) and with GNU C compiler (`gcc`) too. A parallel code perfomed the same using UPC approach (`upcc` compiler). We can see the profiling results in Figure 4.1.

Figure 4.1 shows the results obtained with the three compilers. One can see that the Intel C compiler gave the best results. The GNU C compiled code was *ten times* slower than the `icc` compiler. For this reason, Berkeley UPC was linked with Intel C compiler to compile C code. The same code was written using UPC approach. The result of the UPC benchmark was plotted with blue bars. One can see that the local variables were approximately the same fast (on a single core) as in the `icc` case. Using shared variables were *hundred times* slower than the local variables. This shows the need of locality exploitation in any code, i.e. *all the expensive processes repeated many times shall be carried out using local variables.*

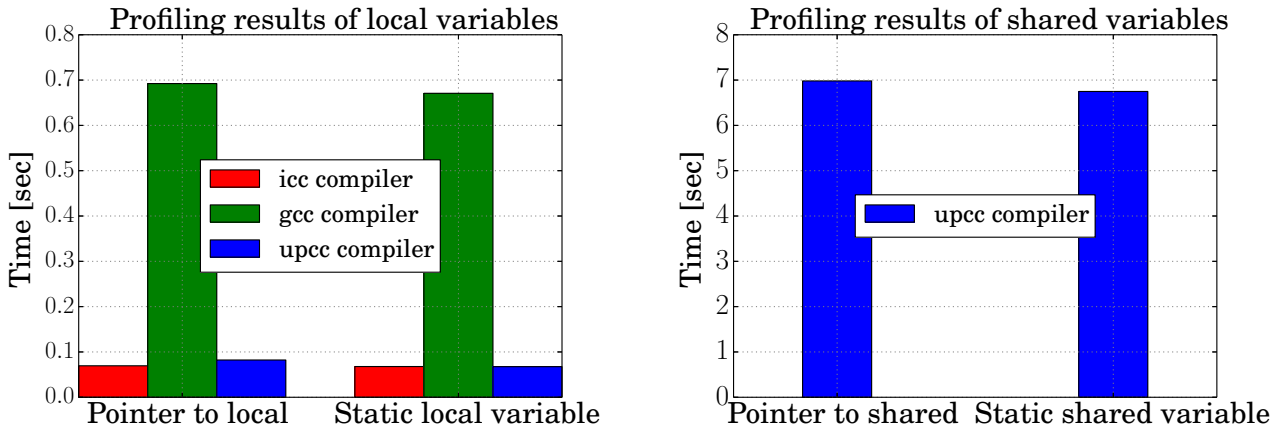



**Figure 4.1.** Benchmark of different data types: execution time using `icc`, `gcc` and `upcc` compilers

A solver for one-dimensional heat conduction problem was implemented in serial and parallel. Two kind of parallel codes were created: shared and local variable based ones. The computational time was measured and the speed-up compared to the serial code was plotted in Figure 4.2. The speed-up was defined as the ratio of the serial and the parallel codes execution time ($SU = t_{serial}/t_{parallel}$). One can see that the speed-up was better for the local variables. The difference between the speed-up became more significant as the number of threads increased. One shall note that *involving more threads does not necessarily overcome the overhead caused by the use of shared memory.*

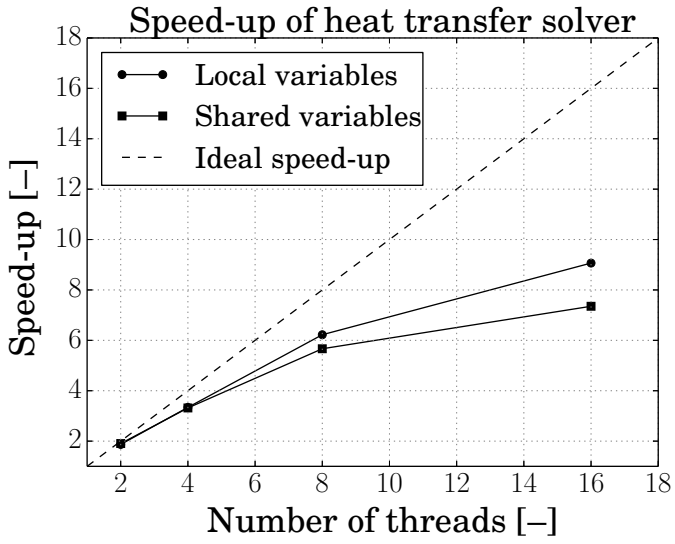



**Figure 4.2.** Benchmark of one dimensional heat transfer solver

## 4.2 Code Reformulation

There were two codes available for parallelisation. Both of them was written in C++. The first step was to reformulate them to C. This was a non-trivial step, since one has different opportunities in C++, which does not exist in C (objects). These properties of the formerly existing codes had to be "transformed" to C. First of all let us consider the properties of the existing codes.

The two available solvers had their own advantages and disadvantages. The code developed by Teschner (Approach A) [2] was a well structured, consistent, massively object-oriented C++ code. The solver written by Abbruzzese (Approach B) [9] was an easy-to-read and also well structured C++ code.

| **Approach A** | **Approach B** |
|---|---|
| User friendly interface: menu with help or read an ini file. | Need to modify the properties in the code. Need to compile every time. |
| It can not be run remotely. | It can be run on cluster. |
| Massive, long and complex code. | Shorter code. |
| Mesh is created in ANSYS ICEM. | Mesh created in Pointwise and with own mesher. |
| Exact representation of boundaries | Not accurate boundaries. |
| Mesh cannot be further modified during run-time. | Mesh can be further modified during run-time. |

**Table 4.1.** Comparison of the available codes

Table 4.1 shows the advantages and disadvantages of the two available codes compared to each other. Approach A is a massively object-oriented code, which has a *user friendly interface* (within the terminal). The interface offers to set e.g. the boundary conditions separately on each boundary. The set-up can be also performed by using ini file, which location has to be an input for the solver via the terminal based interface. One can see that this implies the need of user at runtime, which means that this code *can not be run remotely* on high performance computer (HPC). Opposite to this, Approach B was developed such that the input properties have to be implemented directly to the code, i.e. it has to be compiled before each run. This means that the code can be executed on an HPC system, which is required to use several CPUs. This is a must feature for the new code. The drawback of Approach B is the compilation before every simulation (low portability).

The main difference between the solvers is in the meshing approach. This is also followed by other differences, which are inherited to the different meshes. The former solver requires a mesh created in ANSYS ICEM and saved to CGNS, which is a binary

file. This mesh is then can be loaded to the solver. The other solver requires a geometry layout (sketch) created in Pointwise. Based on this layout, another C++ program creates the final uniform mesh. The solver imports the created files and further process them. The boundary conditions are defined at the meshing stage in this approach. Every cell has a boolean type property, which decides whether the cell is fluid or not. Once it is fluid, calculations are performed on the given cell. In Approach B, a uniform mesh is created i.e. if the boundaries are curved, they are resolved as steps (cells are equal in size). This can be seen in Figure 4.3, where a cylinder meshed with the latter approach is shown. The boundaries in the former is exact, i.e. only the fluid part is meshed like in conventional CFD approaches. In case of calculating e.g. multiphase flow or sedimentation using Approach B, one can change the type of cells easily in the code during runtime (fluid → solid or solid → fluid).

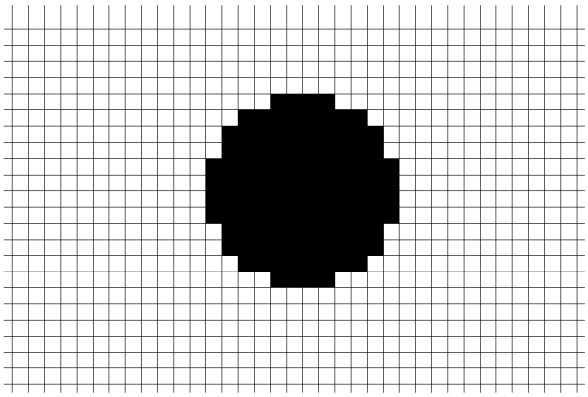


**Figure 4.3.** Mesh of a cylinder using Approach B, white: fluid, black: solid cells

### 4.2.1 The Reformulated C Code

From the available in-house codes [2, 9] a new code was written in the collaboration with Józsa [46]. This step meant that from the available C++ codes, a new C code had to be formulated due to portability reasons (C offers better portability than C++). The new code intend to unify the advantages of the former codes, and it has some new features. The code uses uniform mesh and operates with fluid/solid cells. Figure 4.4 shows the flowchart of the new C code.

The new code offers the opportunity to run it remotely on cluster. The program reads a mesh from a given place and it imports the set-up properties from an .ini file. This means that there is no need of compilation any more. It writes all the results to a folder within the working directory ("Results"). The solver properties i.e. which collision model to use, how often to save, velocity and viscosity etc. can be easily adjusted via an ini file (SetUpData.ini).

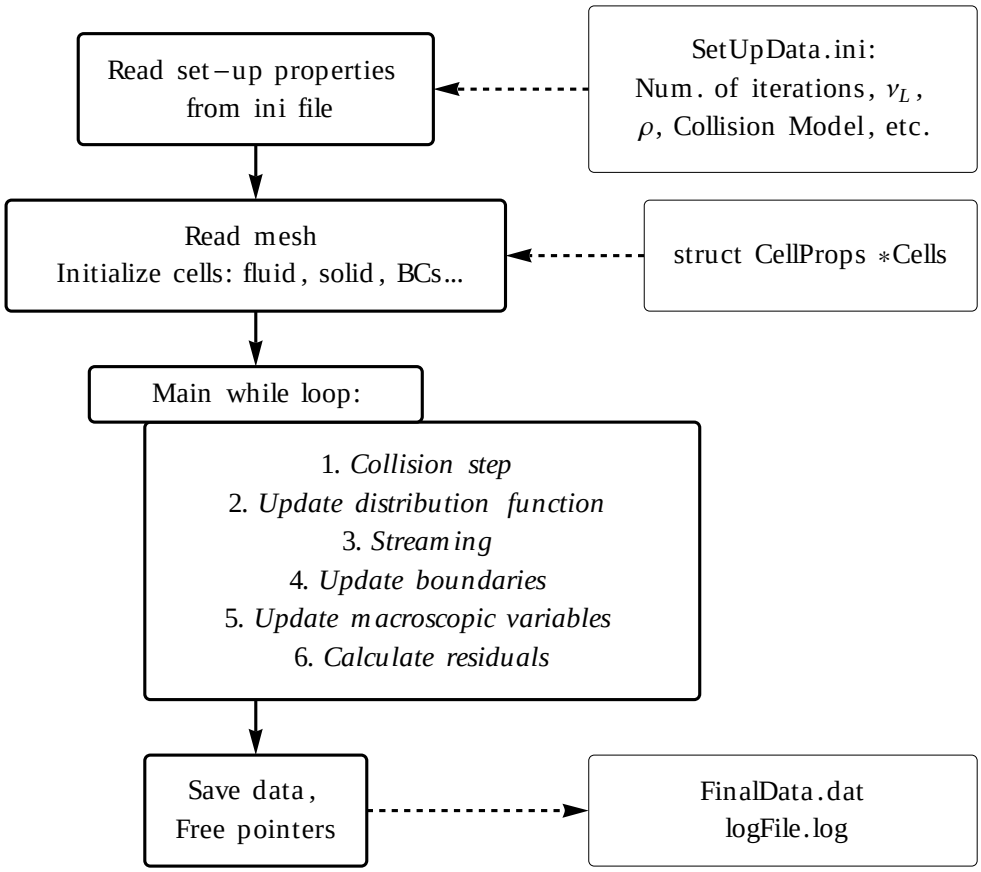



**Figure 4.4.** Flowchart of the reformulated code

The flowchart of the new code can be seen in Figure 4.4. After reading the ini file, the solver reads the mesh. At this stage, raw mesh data is available. A `struct` was defined in C, which includes all the properties of a cell (boundary type, Cell ID, $x$ and $y$ coordinate, solid or fluid cell etc.). These properties are defined at the initialization step. This part is a crucial step in the solver, since in the future calculations, the code will rely on the informations defined at this step. The core of the solver is executed next, where the iterative process takes place (main while loop). This process runs until the formerly defined (ini file) number of iterations are performed. Auto-saved data is written occasionally after the predefined number of iterations (ini file). The residuals are calculated on the fly, which are also written in a log file. When the iterative process ends, the solver quits and saves the final state and finally the memory is freed.

One can see that the developed code can be executed on an HPC system. It needs to be compiled only once, and the set-up can be done via an ini file for each case one wish to run. Based on this, one can see that the developed code forms a good basis for parallelisation.

The original C++ code used a pointer to pointer to represent matrices. For achieving good performance in C [47], this had to be transformed to vector form, i.e. a single pointer was responsible to store the properties of the whole mesh. The change of syntax assuming a mesh size $n \times m$ in $x$ and $y$ directions is shown below, where in each cells the $x$ velocity component is set to zero:

(a) Pointer to pointer representation:

```
// Structure for the cell
   properties
struct CellProps **Cells;

// Allocate:
Cells = calloc(n, sizeof(struct
   CellProps*));
for (i = 0; i < n; i++)
  Cells[i] = calloc(m, sizeof(
   struct CellProps));

// Set U velocity to zero
for (i=0; i<m;i++){
    for (j=0; j<n;j++){
      Cells[j][i].U = 0;}
}
```

(b) Single vector representation of matrix in the reformulated code:

```
// Structure for the cell
   properties
struct CellProps *Cells;

// Allocate:
Cells = calloc((n*m), sizeof(
   struct CellProps));


// Set U velocity to zero
for(j=0; j<m; j++){
  for(i=0; i<n; i++;){
      (Cells+j*n+i)->U = 0;}
}
```

One wish to note that although the change might seems simple from syntax side, *accessing neighbour elements is much different* in the latter representation. This difficulty rises when the streaming process occur. In this case, all the eight neighbouring elements have to be accessed. This means shifting in $x$ and $y$ directions at the same time.

The performance gain of changing from matrix to vector representation was evaluated. The lid driven cavity validation case (see at the end of this section) was used for benchmarking the codes. The mesh consisted of 640k cells. The gained speed-up of converting the code from matrix (pointer to pointer) representation to vector (single pointer) gained 40 [%] speed-up using only single thread.

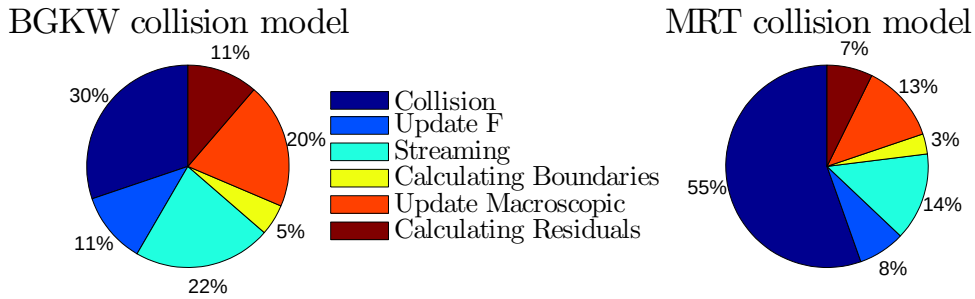



**Figure 4.5.** Profiling results of the reformulated serial C code

Finally, the profiling of the reformulated code was carried out. The required time for each part of the solver was measured starting from the initialization process. A pie chart in Figure 4.5 shows the percentage of how long did each step take. On the left hand side, one can see the results gained with the BGKW collision model, while on the right hand side the MRT collision model was plotted. The first important difference is that the MRT collision model was almost *twice* more expensive to calculate than the BGKW collision model. This was expected during the discussion of methodology, because MRT collision model uses matrix multiplication to model the collision step. Opposite to this, BGKW is a single equation model. In both cases, the most expensive step was the collision step, which is a totally parallelisable step (independent from neighbours). The second most expensive part in a for loop was the streaming and the third one was the calculation of macroscopic variables.

## 4.3 Parallelisation

In this sub-section the parallelisation of the reformulated C code will be discussed. One also finds the gained speed-up results within the following figures.

### 4.3.1 Parallelisation: Shared Approach

Since Unified Parallel C offers the usage of shared address space, the first step toward parallelisation was the usage of the shared memory. This has a similar layout as in OpenMP, but in this case, each variable has affinity to a certain thread. Nota bene: This is the main novelty of UPC compared to the conventional approaches. The UPC extension of C language offers to use a special `for` loop. This assigns the calculation task of each loop to a certain thread, based on the affinity. This means that by using `upc_forall`, it is relatively *easy* and *quick* to use shared memory, once the user has the serial code. For example transferring a double `for` loop from serial to parallel reads as follows:

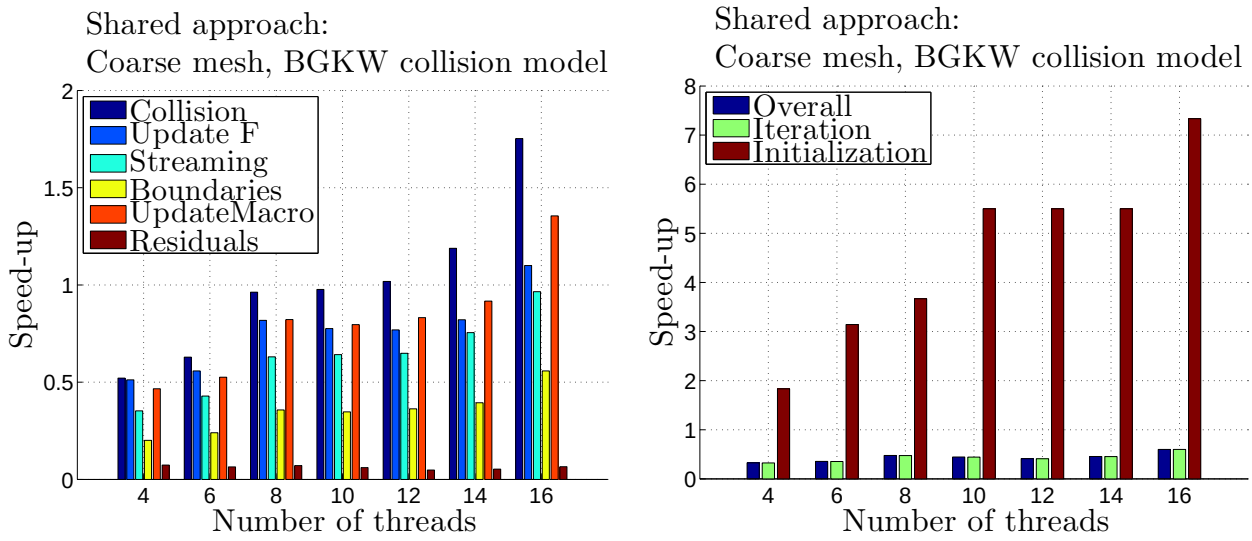
(a) serial code

```
// Create structure for the cell
   properties
struct CellProps *Cells;

// Allocate memory
Cells = calloc((n*m),sizeof(struct
    CellProps));

// Fill x velocity with zeros
for(j=0; j<m; j++){
  for(i=0; i<n; i++;){
      (Cells+j*n+i)->U = 0;
  }
}
```

(b) parallel using shared variables

```
// Create shared structure for the
    cell properties
shared [BLOCKSIZE] struct
   CellProps *Cells;

// Allocate memory on each thread
Cells = (shared [BLOCKSIZE] struct
    CellProps*) upc_all_alloc(
   THREADS, BLOCKSIZE*sizeof(
   struct CellProps));

// Fill x velocity with zeros
for(j=0; j<m; j++){
  upc_forall(i=0; i<n; i++; &Cells
   [j*n+i]){
      (Cells+j*n+i)->U = 0;
  }
}
```

In the parallel code, the second line declares a shared pointer to a struct, with a certain `BLOCKSIZE`. The block size tells the compiler that *how many cells* belong to each thread. This can be done for example by making the block size equal to `n*m/THREADS` (knowing that the mesh has a size of $n \times m$). The `THREADS` keyword in UPC is the number of threads on which the solver is executed. Although this approach seems




(a) Speed-up of the Iterative Steps

(b) Main Speed-up Results

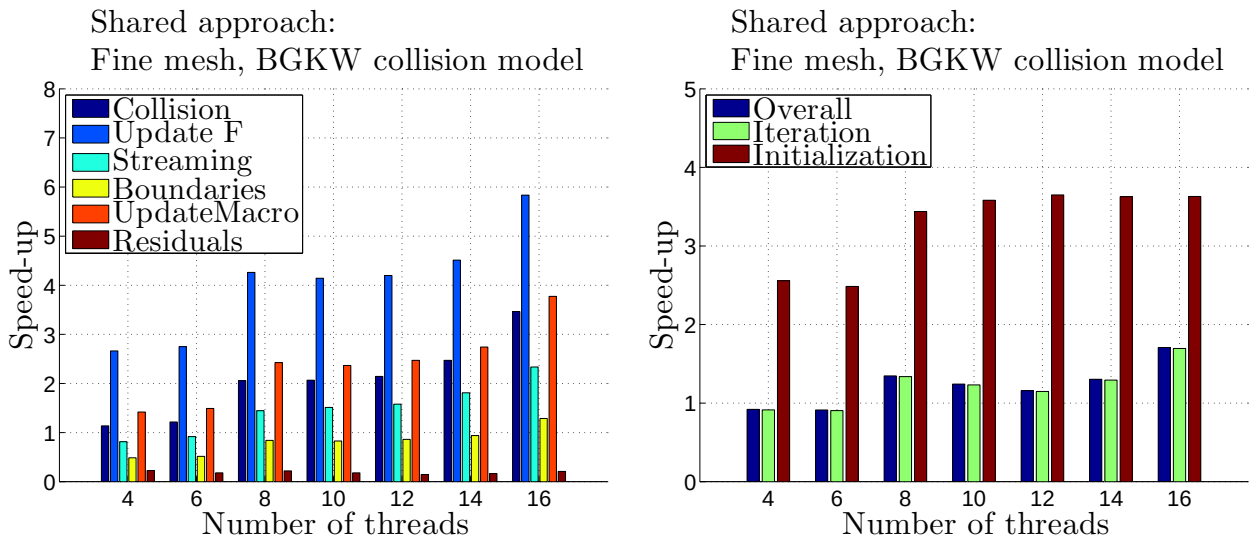
**Figure 4.6.** Speed-up results for the coarse mesh using shared approach. BGKW collision model ran on 4 to 16 threads (left: speed-up of the different steps within the iterative loop, right: overall process, iterative process and initialization process speed-up)

relatively straight forward and easy, note that in this case *the block size needs to be an integer*. This means that `n*m` must be divisible by the number of threads. This restriction can be satisfied by creating an appropriate mesh. A higher restriction for this approach is given by UPC itself, which requires the programmer to make the block size *compile time constant*. This means that *before* compiling the code one needs to know the size of the mesh. Therefore, one needs a different executable for each mesh. This is the *main drawback* of this approach. Obviously, this highly decreases the portability of the code. The restriction goes against one of the main reasons why the code was translated from C++ to C (better portability).

The allocation is done in line 5, where each thread allocates its own portion (`BLOCKSIZE` number of cells) with using the function `upc_all_alloc`. In the calculation process, the main difference from the serial code is that in the inner loop the `upc_forall` function is used instead of the standard `for` function. In this case, the additional fourth term is the affinity term (see Section 2). This decides which thread will perform the calculation (to which the $\mathtt{i}^{\mathrm{th}}$ element has its affinity).

The speed-up of the parallelised code was tested on Cranfield University's Astral HPC system using five different meshes. The speed-up was defined as follows: $SU = t_{serial}/t_{parallel}$. The simulated flow was the lid-driven cavity with a Reynolds number of 1000 (see the validation at the end of this section). The mesh sizes were as follows: 100×100 (coarse), 200×200 (medium), 400×400 (fine), 800×800 (ultrafine) and 1600×1600 (hyperfine). This means that the smallest mesh consisted of 10,000

cells, while the largest had 2.56 million cells. In the followings, the gained speed-ups will be discussed.




(a) Speed-up of the Iterative Steps

(b) Main Speed-up Results

**Figure 4.7.** Speed-up results for the fine mesh using shared approach, BGKW collision model ran on 4 to 16 threads (left: speed-up of the different steps within the iterative loop, right: overall process, iterative process and initialization process speed-up)

Figure 4.6 shows the speed-up bar charts earned by using shared memory approach on the coarse mesh (100×100 cells). Figure 4.7 shows the same results but earned with the fine mesh (400×400 cells). On the left hand side of both figures, one finds the different speed-up results of the steps within the iterative loop. On the right hand side the overall speed-up (whole execution), the speed-up of the iterative process and the speed-ip of the initialization processes were plotted. The first and the most significant result, which can be seen on the plots is that *almost no* speed-up was gained with this approach. This has two reasons: (a) using shared variables is usually slower than local variables (this is one of the main reason why MPI became widely used) and (b) Astral (Intel based architecture) *does not* support from hardware side the handling and addressing of shared variables, while for example Cray architecture has such support [31]. These yield to the very poor speed-up results. We can see that for the coarse mesh no speed-up was gained at all, while for the fine mesh using 16 threads was only slightly faster than the serial code on a single thread. Also note that this speed-up was gained mainly via the the speed-up of the initialization process. For this reason, no cross node simulations were performed using such approach. One can see that on Intel architecture, one needs to *use local variables* more extensively.

### 4.3.2 Parallelisation: Local Approach

As we saw previously, the shared memory approach helps the user to create a parallel program quickly, but might not ends up with a portable and well performing code. As a next step, local data was used. This meant that MPI like approach of the problem was necessary. Programming such layout in UPC still requires less conceptual effort than carrying out the same task in MPI [16]. A few major modifications were necessary compared to the shared approach. In this case, each thread has an own portion of the `struct` called `Cells`. This means that *communication between the threads* became necessary. This can be done via the shared memory space. Let us see how this approach affects the major steps.

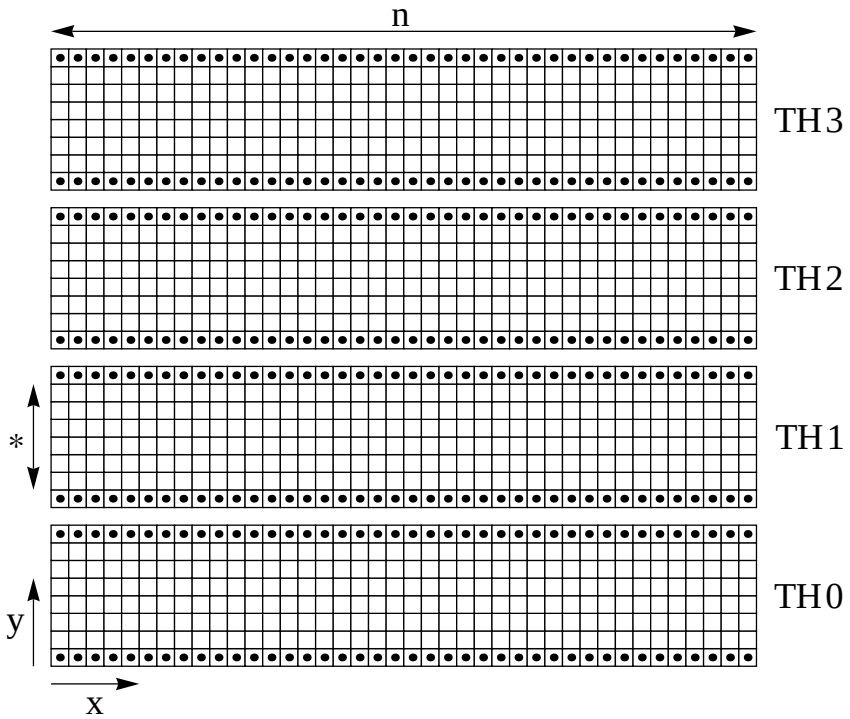



**Figure 4.8.** Distribution of cells among the threads (marked cells: ghost cells, hollow cells: calculated cells)

First of all note that in this case the opportunity was given to create an executable, which can read any kind of meshes, since the mesh was not stored in the shared space any more (no compile time restriction). The distribution of the load was as follows. Each thread received a portion based on the following relation: `BLOCKSIZE = n*((int)(m/THREADS+1))` (marked with * in Figure 4.8). This means that the mesh was "sliced" along the $y$ direction as it can be seen in Figure 4.8. The figure shows that the mesh was distributed between the threads by applying straight slices. Therefore,

the communication was easier and uniform on all thread boundaries (see soon). The dotted cells were the ghost cells along the thread boundaries (see Figure 4.8). These will help us to communicate between the threads (discussed in the following paragraphs). We can see that these two cell rows had to be added at the allocation to the mesh size on each thread.

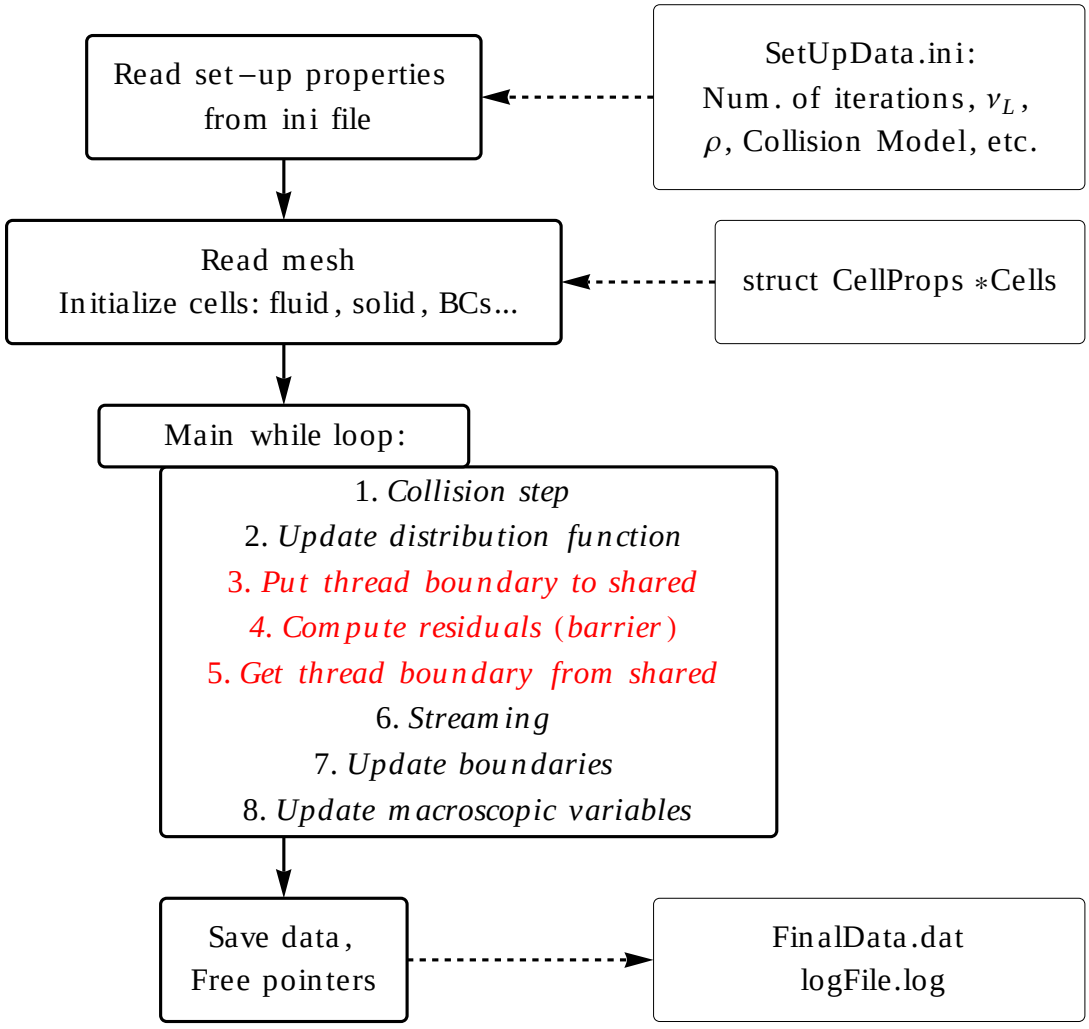



**Figure 4.9.** Flowchart of the local variables based parallel code

In the followings, the changes compared to the shared parallel code will be discussed. Figure 4.9 shows the flowchart of the local variables based parallel code. The figure will help us to follow the progress which takes place in this case.

The first step was to read the ini file and the mesh. This was *done by each thread*.

The second step was to initialize the cells, but in this case, because of the additional ghost cells, the indices had to be shifted along the $y$ direction. This was necessary to

distribute the information properly between the threads. Once the initialization was done, the pointers which store the raw mesh were freed.

The iterative process began. The first step was the collision step, which was totally parallel and no synchronisation was needed between the threads. Note that the `for` loops in the main loop act only on the active cells, i.e. the ghost cells were not used. Nota bene: The mesh size was `n`×`m` in $x$ and $y$ directions. For this purpose, the outer `for` loop acts only on the appropriate cells (`j = 1 ... m/THREADS`). See a sample C code below (page 41). After the collision step, the distribution function was updated. Before we stream the updated distribution function, we shall perform the communication between the threads. This communication is shown in details in Figure 4.10, and Figure 4.9 marks this step with red letters. Figure 4.10 shows the "cross section" of the mesh along the $y$ direction (at arbitrary $x$ coordinate). We can see the case when 4 threads are communicating with each other through the shared space. The shared cells were plotted with red dots, while the ghost cells were marked green. The black dots show the active cells, on which the calculations were performed. The others were required only for communication purposes. The blue arrows show us the `upc_memput()` process, which writes local data to the shared memory. The grey arrows show the `upc_memget()` process, when the data is copied from the shared space to the local memory. Between the putting and getting processes we need to synchronise the threads to have the proper information ready. This synchronisation barrier was aligned (same barrier) with the residual calculation barrier, i.e. between the `upc_memput()` and `upc_memget()`, the code performed the residual calculation (see Figure 4.9). This also required a barrier (see soon). The applied flowchart offers the compiler the opportunity to overlap the two processes. Note that the more barrier is introduced in the code, the less speed-up can be gained. In the final code only *one* barrier was used (the formerly mentioned one). This means that the number of barriers was minimised, since one requires at least one barrier in the iterative loop.

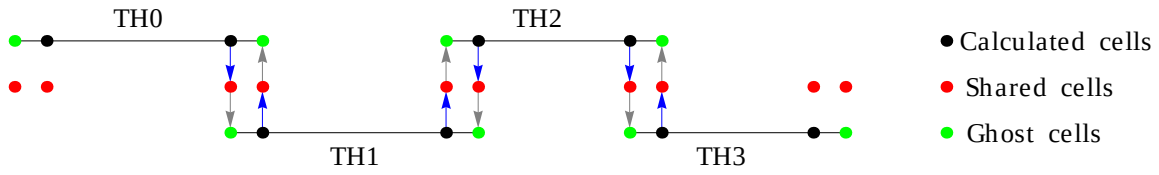



**Figure 4.10.** Distribution of cells among the threads

(a) parallel using *shared* variables

```
// Create shared structure for the
    cell properties
shared [BLOCKSIZE] struct
   CellProps *Cells;

// Allocate memory on each thread
Cells = (shared [BLOCKSIZE] struct
    CellProps*) upc_all_alloc(
   THREADS, BLOCKSIZE*sizeof(
   struct CellProps));

// Fill x velocity with zeros
for(j=0; j<m; j++){
  upc_forall(i=0; i<n; i++; &Cells
   [j*n+i]){
      (Cells+j*n+i)->U = 0;
  }
}
```

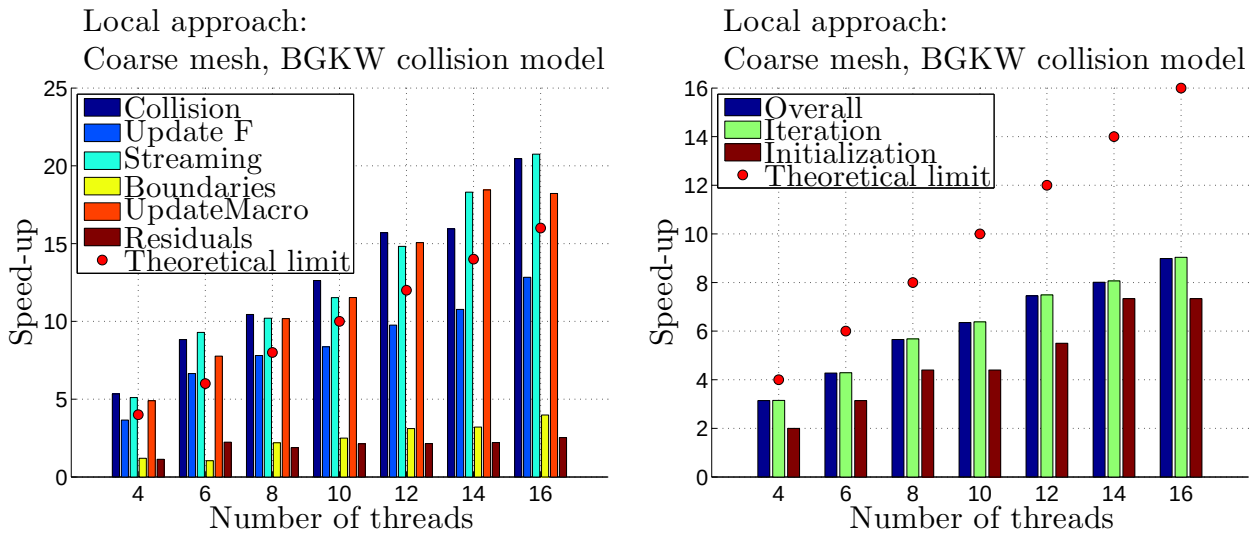
(b) parallel using *local* variables

```
// Create local structure for the
   cell properties
struct CellProps *Cells;

// Allocate memory on each thread
Cells = malloc((m/THREADS+2)*n*
   sizeof(struct CellProps));



// Fill x velocity with zeros
for(j=1; j<(m/THREADS)+1; j++){
  for(i=0; i<n; i++){
      (Cells+j*n+i)->U = 0;
  }
}
```

As it was noted, the residual calculation takes place between the data putting and getting processes (see Figure 4.9). We have seen in Section 3 that the main task from programming point of view during the residual calculation is a *summation*. This means that e.g. the density must be summed along all the cells, but none of the threads has the required number of cells available at once. This problem was resolved by summing the density on each thread separately using local variables (sub summations). This data is then written to a shared variable (sub summation results). Finally, the first thread summed up this data. The number of elements of this vector is equal to the number of threads (for example for summing the density). Before further adding up the sub summation results, one needs to use a *barrier* to ensure that all the sub sum results has arrived to the shared space. The final step was to calculate the change of sums and write the result to a file.

The communication was followed by the streaming process. After this, the boundaries were treated and finally the macroscopic variables were calculated.

The second parallelisation approach was tested with the same meshes on the same architecture as in the shared approach (see on page 37). We can see in Figures 4.11 and 4.12 that in this case the code showed significant speed-up. Note that the theoretical speed-up limit is indicated in the figure. Using a given number of threads ($N$), the theoretical limit for speed-up is generally defined as $SU = N$ (direct proportionality). The same two plots were made as formerly to plot the gained performance. The ideal speed-up was plotted with red circles, which show us the limit of speed-up which can



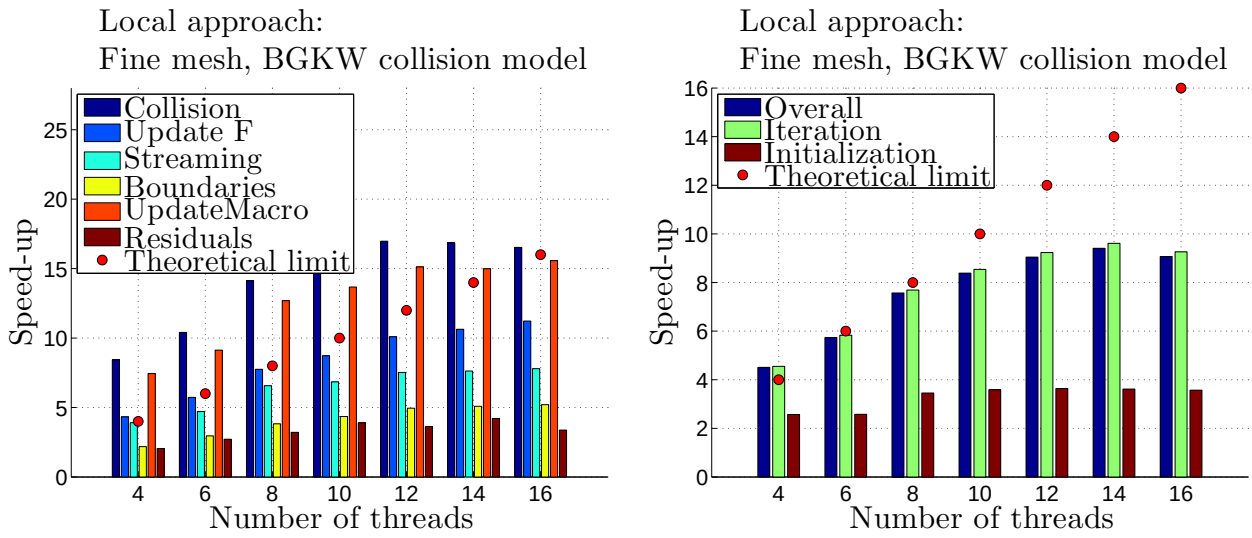

(a) Speed-up of the Iterative Steps

(b) Main Speed-up Results

**Figure 4.11.** Speed-up results for the coarse mesh using local approach, BGKW collision model ran on 4 to 16 threads (left: speed-up of the different steps within the iterative loop, right: overall process, iterative process and initialization process speed-up)

be maximally reached using the classical methods (MPI, OpenMP). It can be seen that the theoretical limit was exceeded by multiple steps for both meshes within the iterative loop. This was because the compiler compiled the code in such a way that during the execution some processes were *overlapped*. The best performing step was the collision step, since this is a fully parallel step. The second and third best speed-up was achieved during the update of macroscopic variables and the update of the distribution function (both totally scalable). Note that the worst performing step was the residual calculation, which requires a few calculations using shared variables. This step was the main limitation of gaining further speed-up. The reasons for this is the same as in the formerly seen shared approach (architecture and shared memory usage). In terms of the overall performance, we can state that better performance was achieved using fine mesh, since in this case the threads had "more job" to do in ratio compared to the required communication. In the fine mesh case, the ideal speed-up was also exceeded by the code in case of using 4 threads.

As we can see in Figure 4.13, using the MRT collision model, which is more expensive than the BGKW collision model, gives better results. For the fine mesh, the speed-up using BGKW collision model was above the ideal for using 4 threads only, while for the MRT collision model the *speed-up exceeded the ideal speed-up up to 10 threads* for the same mesh. We can see that "keeping the CPU busy" gives the compiler more opportunity to hide latency between the different steps. In terms of the iterative steps, the MRT was also better. The collision step was almost twice faster (using 16 threads)



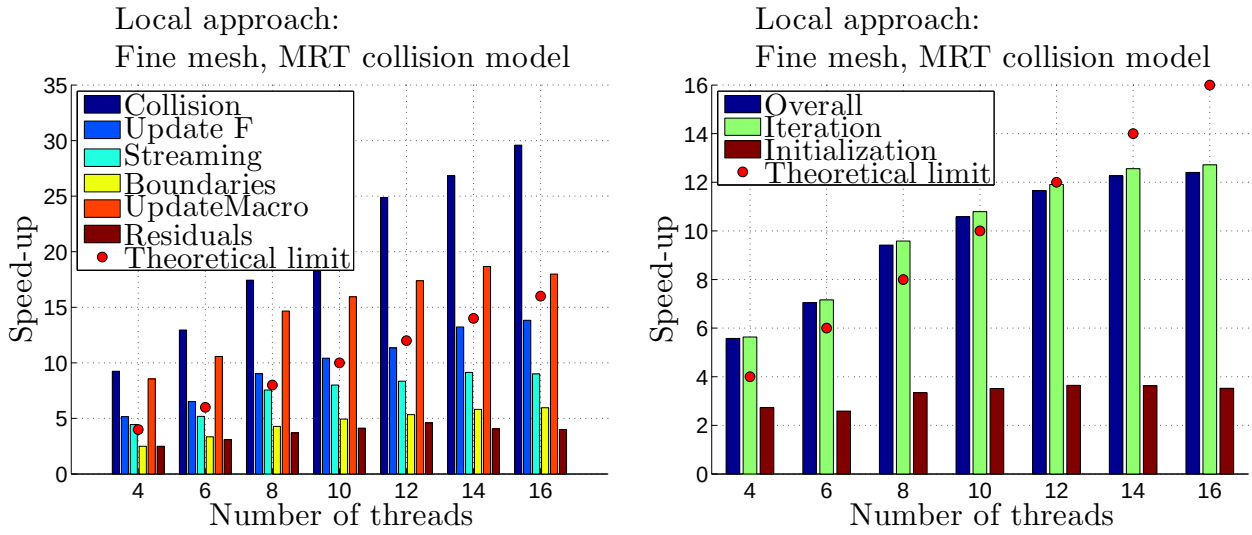

(a) Speed-up of the Iterative Steps

(b) Main Speed-up Results

**Figure 4.12.** Speed-up results for the fine mesh using local approach, BGKW collision model ran on 4 to 16 threads (left: speed-up of the different steps within the iterative loop, right: overall process, iterative process and initialization process speed-up)

with the MRT collision model. This is not surprising after taking a look at the overall speed-up, since this is the process which is the easiest to parallelise and MRT requires more computation. This means that the compiler could hide the latency mainly at this step. Based on the former few figures (Figures 4.12 and 4.13), one can see that it is worth to further increase the number of threads, and investigate the performance of the code going cross-node.

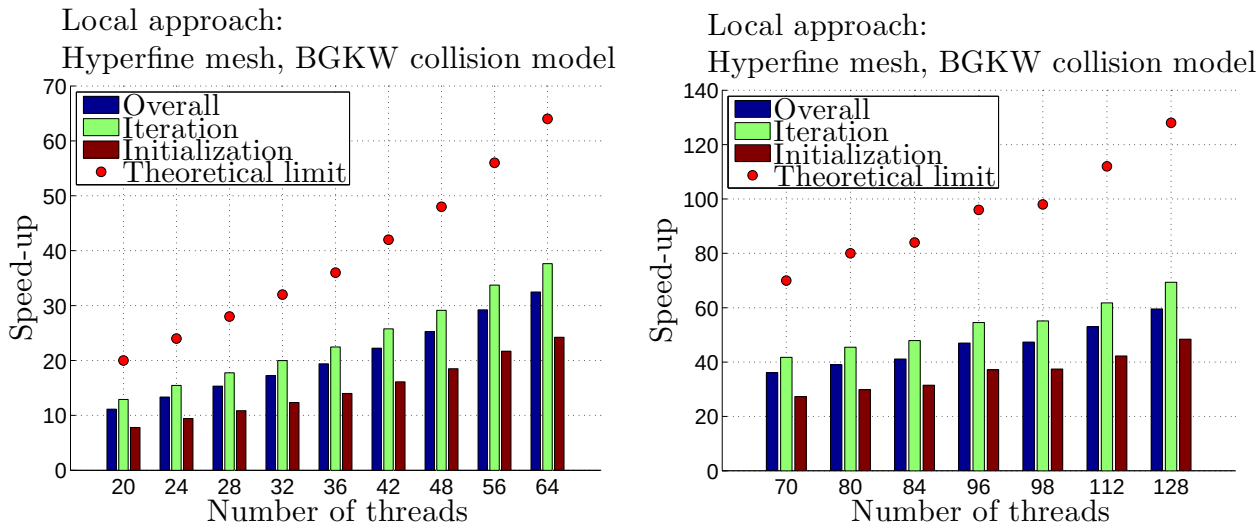
The next step was to evaluate the performance using multiple nodes. This means that until now, the CPUs did not have to use InfiniBand to communicate with each other. When several nodes are involved in the communication, the main advantage of the HPC system is used, which is the connection between several nodes. Roughly speaking using one node in case of Astral offers approximately the computing capability of a workstation. Astral allows the user to use 16 threads within one node. In Figure 4.14, one finds the speed-up results using 20 to 128 processes for the hyperfine mesh (2.56 million cells) beside the BGKW collision model. For example between 20 to 32 threads two nodes were used, between 32 and 48 three nodes and so on. The highest number of nodes was 8. We can see in the figures that the cross node execution was successful, i.e. the code showed continuous speed-up using several nodes as well. Running 2.56 million cells using 128 nodes means *low load* for such high number of CPUs. The performance met the expectations: For such number of threads, one usually uses far bigger meshes (tens of million cells). On the other hand, our solver is a 2D solver, which means that using too fine mesh is pointless after a certain mesh size.




(a) Speed-up of the Iterative Steps

(b) Main Speed-up Results

**Figure 4.13.** Speed-up results for the fine mesh using local approach, MRT collision model ran on 4 to 16 threads (left: speed-up of the different steps within the iterative loop, right: overall process, iterative process and initialization process speed-up)



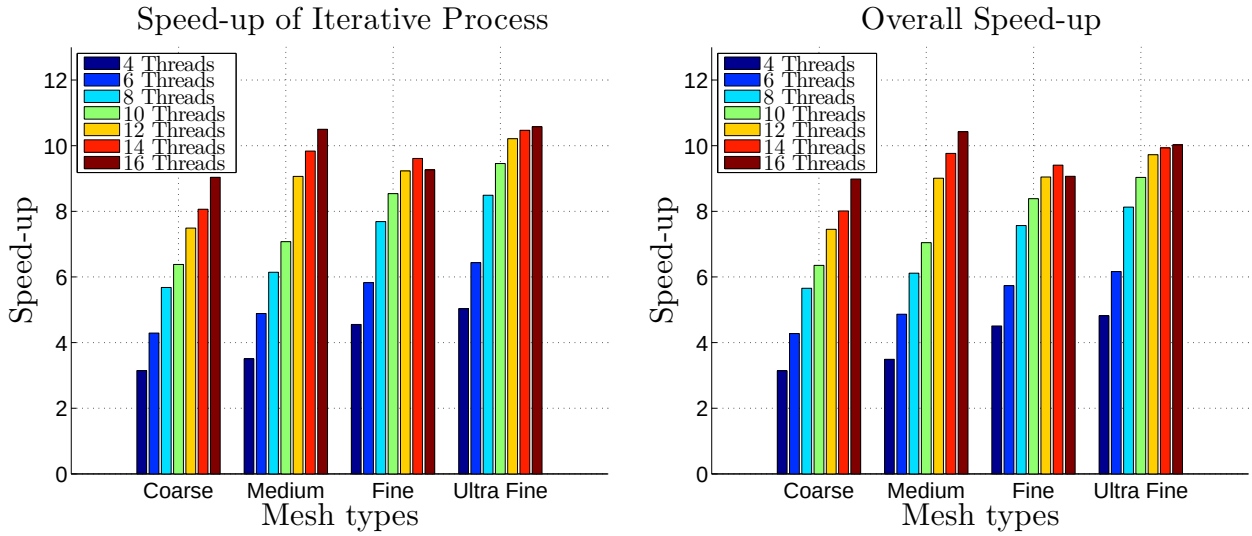

(a) Main Speed-up Results: 20 to 64 threads

(b) Main Speed-up Results: 70 to 128 threads

**Figure 4.14.** Speed-up results for the fine mesh using local approach, BGKW collision model ran on 20 to 128 threads

This is because 2D solvers are also used because of saving computational time. These limitations can be well seen in Figure 4.14. The efficiency of the code was far below the ideal speed-up because of the low load. The speed-up was linear throughout all the nodes which shows that applying bigger computational load on the CPUs would mean better speed-up. The final conclusion of this investigation is that the code works

efficiently using several nodes and in case of having bigger meshes, the speed-up charts would show better quantitative results.




(a) Speed-up of the Iterative Process (b) Overall Speed-up

**Figure 4.15.** Speed-up results for different mesh sizes on a singe node

The last speed-up study was investigating how the performance changes through the load, i.e. how does the code performs using a given number of threads on a given size of mesh. We wish to have better results for bigger meshes, since we have already seen on the cross-node study that the load given to the CPUs was still low. After taking a look at Figures 4.11 and 4.12, we can see that the latter performed better (as it was already discussed before). The comparison of speed-up in terms of different meshes sizes was plotted in Figure 4.15. The figure shows the change in speed-up along the different meshes. The figure meets the expectations. The bigger the mesh was, the better speed-up was achieved. The number of cells was always four times bigger as we go from coarser to finer mesh. The achieved speed-up was approximately linear, especially in case of using lower number of threads. The figure suggests that extending the solver to 3D would be a reasonable step because in that case the load would be more significant on the CPUs.

Figure 4.16 shows how expensive the different steps were during the iterative process. The graph compares all the three codes discussed before. We can see that moving from the serial code to the local code only some changes occurred. This means that the distribution of work remains approximately the same for most of the steps. The biggest change was experienced in the collision step, which took almost half the time to perform in the local approach (28 [%] $\rightarrow$ 16 [%]). Updating the distribution function (Update F) and the streaming step was almost the same expensive as in the local code.

The boundaries took almost twice the time to handle, while the residual calculation was slightly cheaper. Communication between the threads (ghost cells) took only 1 [%] in a loop. The figure shows the work distribution for the fine mesh, therefore we can say that the communication time was reasonably low. Note that the pie charts show an average through the used number of threads. The time of the iterative loop was divided by the number of iterations. This calculation was made for different number of threads. This result was averaged and plotted in Figure 4.16.

We can see that the shared code performed much worse than the serial code in this case as well. The most significant difference was that almost half of the time was spent on residual calculation (38 [%]). Note that residual calculation is not a must to perform, and from mathematical point of view, it is a smaller task (compared to other steps). What makes it expensive is the communication it requires. It was performed on the first thread, which summed up all the required variables along all the cells. Based on these results, we can state that the shared approach failed also in terms of exploiting the affinity property offered by UPC and also letting one thread accessing to a whole shared vector.

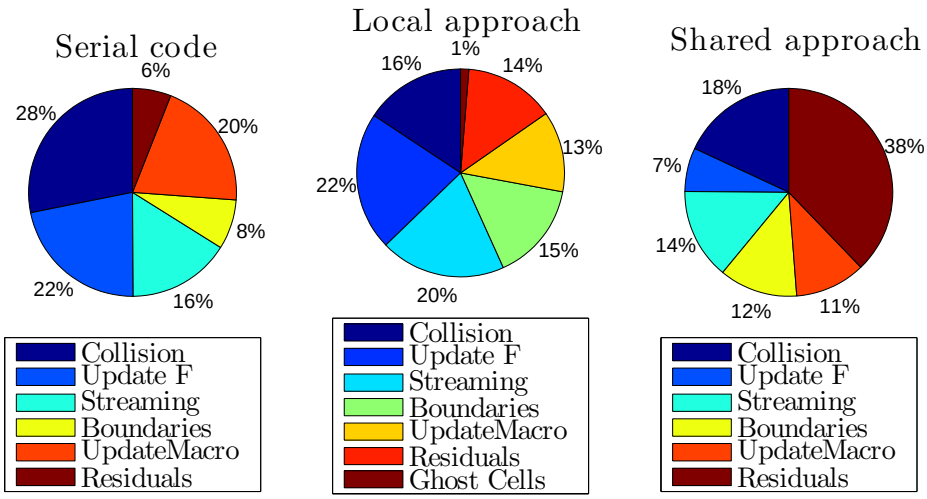



**Figure 4.16.** Pie chart: percentage of time spent on the different steps within a loop during the iterations

In the framework of another thesis, Józsa [46] parallelised the same C code, which was introduced in the beginning of this section. The other study performed the parallelisation in the same year using General Purpose Computing on Graphical Processing Units (GPGPUs). The programming language was CUDA [48], which was executed on the GPGPU cluster of Cranfield University (called Fermi). The cluster consists of several nVidia Tesla C2050 GPU cards. This offers comparison between the results achieved by UPC (ran on Astral) and the results given by CUDA (ran on the Fermi cluster). Józsa found [46] that using double precision on the GPGPU cluster (which is offered only by specific GPUs like Tesla), the speed-up in terms of using 1 million cells was 12. We can see that for such speed-up one needs to use all the threads of a single node on Astral. Bear in mind that single precision is one of the biggest restriction that applies for GPUs. This means that some conventional GPUs (not built for scientific purposes), which are also capable for executing CUDA programs, do not support double precision. In terms of single precision, Józsa reported a speed-up of 16 applying a proper set-up (please find detailed description in [46]). As a conclusion, we can say that comparing the performance of the two approaches, one needs the computing capability of a workstation (approximately) to achieve the same speed-up of a single GPU card.

## 4.4 Validation

The formerly developed codes were validated in [2, 9]. The new reformulated C code and the parallel code (only the local approach, since the shared one did not yield any speed-up) had to be validated. In the followings, the five different validation cases will be discussed and comparison will be made between the two codes. The validation cases were the followings:

- Channel flow (analytical solution available [20]),
- Backward facing step (experiments of Armaly *et al.* [49]),
- Sudden expansion (experimental data from Fearn *et al.* [50]),
- Flow around a cylinder (qualitative study) and
- Lid driven cavity (simulations carried out by Ghia *et al.* and Perumal & Dass [6, 51]).

The used meshes and all the set-up was the same for the parallel and the serial code.

### 4.4.1 Channel Flow

The geometry of the channel can be seen on the left hand side in Figure 4.17. In this case, flow happens between two flat parallel plates. The original flow can be treated as two-dimensional. On the left boundary of the domain, velocity inlet was defined with uniform velocity and on the right boundary we can find outlet. The upper and the lower boundaries were walls (see Figure 4.17). The Reynolds number in this case was

$$Re = \frac{N_{Ch} u_{Ch}}{\nu_L}, \tag{4.1}$$

where $N_{Ch}$ was the number of nodes along the height of the channel. $u_{Ch}$ was the predefined constant velocity at the inlet of the channel. The lattice viscosity ($\nu_L$) was set for 0.01 [−], and the initial density was $\varrho = 5$ [−] for all the cases (also for the latter validation cases). Three meshes were used, which properties were listed in Table 4.2. After calculating the Reynolds number from the given data, one gets that $Re = 100$. Each mesh was a uniform mesh (as for all other validation cases too).

For this validation case, analytical solution was available. Omitting the derivation, the solution of the velocity field ($u_x = u(y)$ since $u_y = 0$) was a parabola [20]. The analytical solution can be seen on the right hand side in Figure 4.17. One can see that the peak of the velocity is 3/2 of the mean velocity in the middle of the channel.

| | Coarse | Medium | Fine |
|---|---|---|---|
| Grid spacing | 0.25 | 0.125 | 0.0625 |
| $x$ velocity ($u_{Ch}$) | 0.05 | 0.025 | 0.0125 |
| $N_x$ | 400 | 800 | 1600 |
| $N_y = N_{Ch}$ | 20 | 40 | 80 |

**Table 4.2.** The different meshes for the channel flow

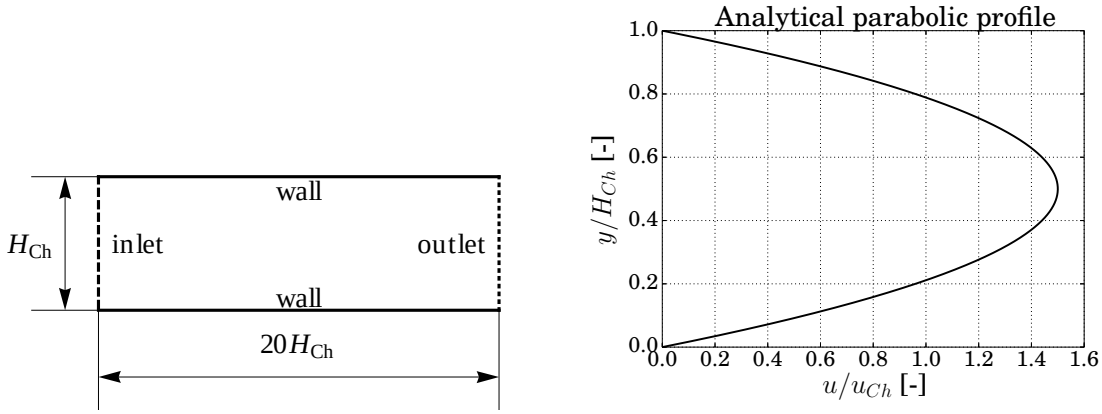



**Figure 4.17.** Geometry of the channel flow (left) and the plot of the analytical velocity profile (right)

As it was seen in Figure 4.17, the length of the domain was 20 times its height. This length was inherited from the former theses [2, 9]. Durst *et al.* [52] offered analytical approximation for the necessary length of the channel to have a fully developed velocity profile. This reads as

$$\frac{L}{H_{Ch}} = \left[0.619^{1.6} + (0.0567Re)^{1.6}\right]^{\frac{1}{1.6}} . \tag{4.2}$$

Based on this relation, one could calculate that the necessary length ($L$) to have a fully developed velocity profile was 5.77 times the height of the domain.

First, let us investigate the results from qualitative point of view. Figure 4.18 shows the streamlines and the normalized velocity contours (dividing the velocity magnitude with $u_{Ch}$). The results were earned with BGKW (left) and TRT (right) models using the fine mesh. In this case, the results were plotted only with using the parallel code (see quantitative comparison later). With the BGKW collision model instability was experienced. By taking a look at the left sub-figure in Figure 4.18, we can see the wavy velocity contours at the outlet. This instability was introduced by the second order resolution of the outlet. This lead to diverging results after a certain number of iterations. The problem could be solved either by using different collision model or decreasing the

accuracy of the outlet to first order. The right sub-figure shows the velocity contours and the streamlines at the end of the domain using TRT collision model. For this case, the simulation remained steady and no instabilities were experienced. Finally, it can be seen in both sub-figures that the flow decelerated at the end of the domain. Former studies [2, 9] also reported this phenomenon.

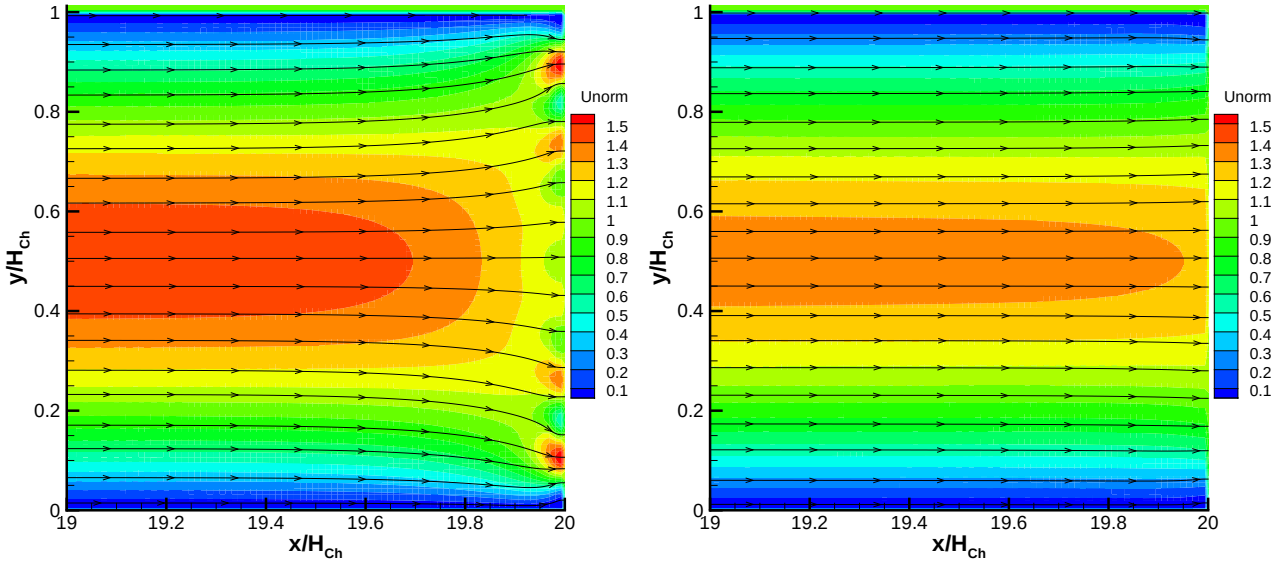



**Figure 4.18.** Streamlines and normalized velocity contours at the end of the domain: BGKW collision model (left) and TRT collision model (right) using fine mesh

The developed dimensionless velocity profiles (after dividing with the mean velocity $u_{Ch}$) were plotted in the middle of the domain ($x/H_{Ch}$=10). The velocity profiles compared to the analytical (dotted) solution were plotted in Figures 4.19 and 4.20. The former one shows the results given by the *serial* code, while the latter one was earned with the *parallel* code. We can see that good accordance was found between the serial and the parallel codes, or with other words the parallel code was resolving the same physics as the serial code did. In case of both figures (serial & parallel), the left sub-figures show that the velocity profile was underestimated by all the three (BGKW, TRT and MRT) collision models. The biggest difference was experienced with the TRT collision model. Note that former studies reported that this collision model performs poorly as it is more likely to diverge [2, 9]. The BGKW collision model showed the best results.

The dimensionless velocity peak along the symmetry axis of the channel was plotted in the right hand side of Figures 4.19 and 4.20. It can be seen that the velocity was increasing along the domain for the BGKW and MRT collision models, while for the MRT collision model it was decreasing after a certain point (around $x/H_{Ch} \approx 5$). We

can see that the parallel and serial results were quite similar to each other in this case too. At the end of the domain all the collision models resulted in decreasing velocity introducing non-physicality in the domain. This means diverging streamlines at the end of the domain. Former studies reported [2, 9] that this effect is originated to the outflow boundary condition, as we have already seen its effect in Figure 4.18. Note that the required length to reach the fully developed velocity profile in the domain met the approximation. The profile developed after $x/H_{Ch} \approx 5$ [−]. The former approximation was 5.77 [−] (see Equation (4.2)).

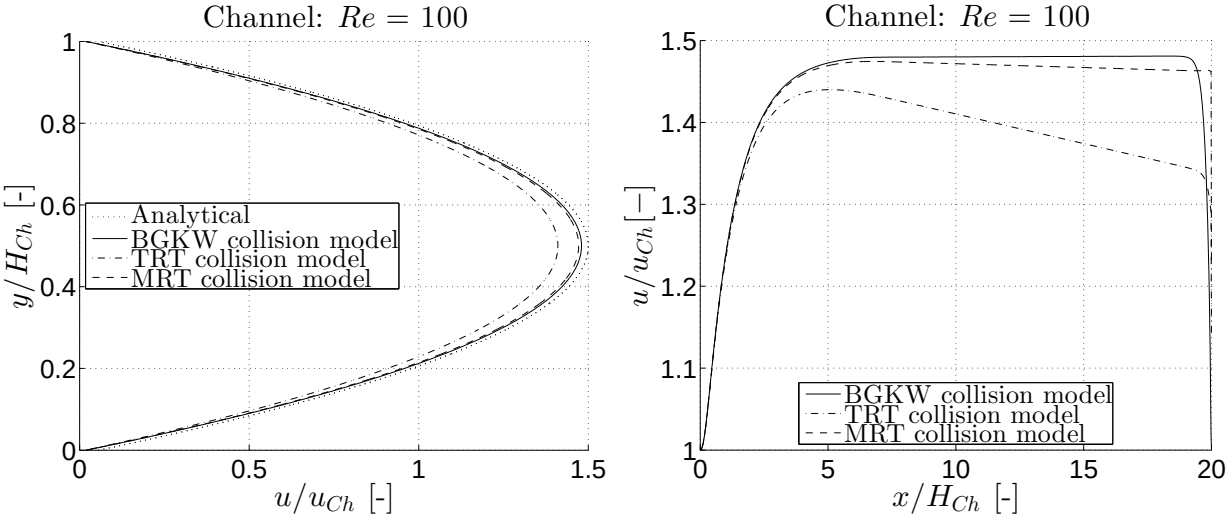



**Figure 4.19.** The developed velocity profiles in the middle of the domain (left) and the axial velocity along the symmetry line (right) given by the *serial* code

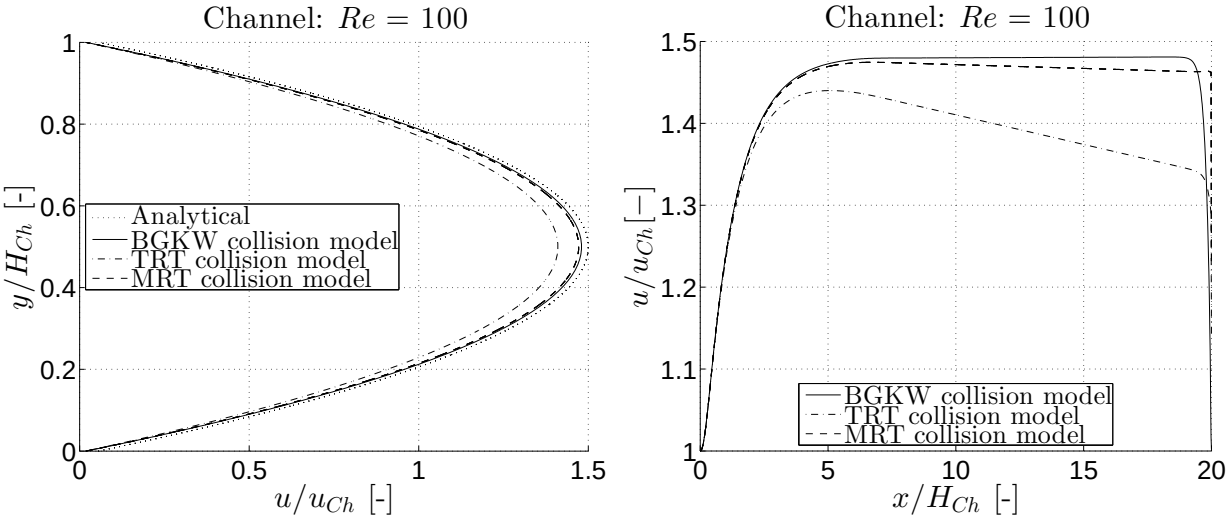



**Figure 4.20.** The developed velocity profiles in the middle of the domain (left) and the axial velocity along the symmetry line (right) given by the *parallel* code

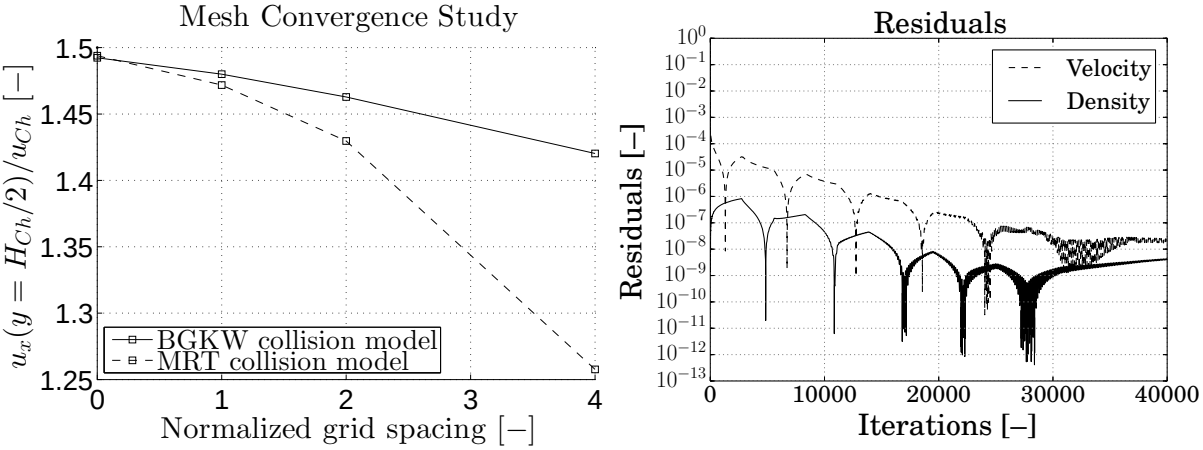



**Figure 4.21.** Mesh convergence study (left) and residuals (right) for the channel flow

Figure 4.21 shows the mesh convergence study (left) and the residuals (right) for the channel flow. The mesh convergence study monitored the dimensionless peak of the velocity at the middle of the domain ($u(x/H_{Ch} = 10, y/H_{Ch} = 0.5)/u_{Ch}$). It was known from the analytical solution that the value of this peak is 1.5 [−]. Table 4.3 shows the calculated order of accuracy of the results, the grid convergence index (GCI) [53] and the extrapolated results of the velocity peak (based on the Richardson extrapolation) [53]. The calculations were based on the velocity at the centre of the domain. The expected order of accuracy was *two*, since the method was second order accurate. This was well resolved for the MRT case, while the TRT gave the worst result. The grid convergence index shows the approximated accuracy of the finest used mesh, which was below 5 [%] for the BGKW and MRT models. The TRT models gave the worst GCI. From the available results, we could extrapolate the velocity peak for infinitely smooth mesh. The table shows these results too. Due to the poor results given by the TRT model, Figure 4.21 shows only the results of the BGKW and MRT models. The $x$ axis shows the normalized grid spacing which was defined in a way that the finest mesh size was divided by the actual mesh size, i.e. the higher this number is, the coarser the mesh is (1 represents the finest used mesh). Zero represents in this axis the infinitely smooth mesh (zero grid spacing). We can see that as the mesh was refined, the results were tending to the analytically calculated value ($u_x = 1.5$ [−]).

| **Collision model** | **Order of Accuracy** (p) | **GCI** [%] | **Extrapolated** $u_x$ |
|---|---|---|---|
| BGKW | 1.305 | 1.17 | 1.494 |
| TRT | 0.582 | 16.4 | 1.595 |
| MRT | 2.031 | 1.47 | 1.492 |

**Table 4.3.** Mesh convergence data for channel flow

On the right hand side of Figure 4.21, one finds the plot of the residuals for the finest mesh using the BGKW collision model. The plot shows the velocity and density residuals. Note that the solver developed in the framework of this thesis was using *double precision* representation of numbers. The residuals reached $\approx 10^{-8}$ after 40,000 iterations. Note that [2] and [9] also experienced the initial wavy behaviour of the residuals, which they reported to be inherent with the lattice Boltzmann method. The reason for this assumed to be that at the beginning of the simulation waves form and they move back and forth in the domain. These waves also left their footprints in the residuals.

### 4.4.2 Backward Facing Step

The next validation case was the backward facing step, which was compared to the experimental results of Armaly *et al.* [49]. In this case, one finds a channel upstream of the domain. From here, the flow enters to a region, where the height of the domain suddenly increases to the twice of the upstream channel height. The geometry can be seen in Figure 4.22. The height of the channel upstream was denoted by $H_{BFS}$. The height of the channel downstream of the step was $2H_{BFS}$. The Reynolds number based on the available experimental results [49] was defined as

$$Re = \frac{u_{BFS} 2N_{H_{BFS}}}{\nu_L}. \tag{4.3}$$

The Reynolds number in the simulations was 100 ($\nu_L$=0.01 in this case too) because experimental data was available for such flow. The origin of the domain was located in the lower left corner of the step.

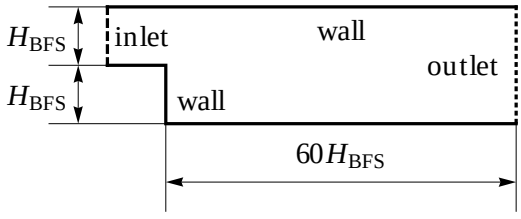



**Figure 4.22.** Geometry of the backward facing step validation case

The streamlines in the domain calculated with the TRT collision model can be seen in Figure 4.23. According to the expectations, a separation zone formed, after which the flow reattached to the lower wall. We can see that the flow decelerates and a vortex forms in the separation zone. Based on the qualitative plot, we can see that the results met the expectations. It is worth to note that, similarly to the channel flow,

the simulations were unstable due to the second order resolution of the outlet. The problem could be solved in the same way as before: using first order method to resolve the outlet. The experimental data showed [49] that for $Re = 100$ the reattachment happens at $x/H_{BFS} = 3.1$. The reattachment length (dimensionless: $l_{reatt}/H_{BFS}$) given by the simulations was listed in Table 4.4. We can see that the best result was given by the MRT collision model, while the TRT model gave the worst result. Note that all the reattachment lengths were underestimated. This meets the former experiences in terms of undershooting the velocities.

| **Collision model** | **Simulation** | **Measurement** | **Error** [%] |
|---|---|---|---|
| BGKW | 2.91 | 3.1 | -6.13 |
| TRT | 2.60 | 3.1 | -16.1 |
| MRT | 2.93 | 3.1 | -5.48 |

**Table 4.4.** Reattachment length of vortices for backward facing step

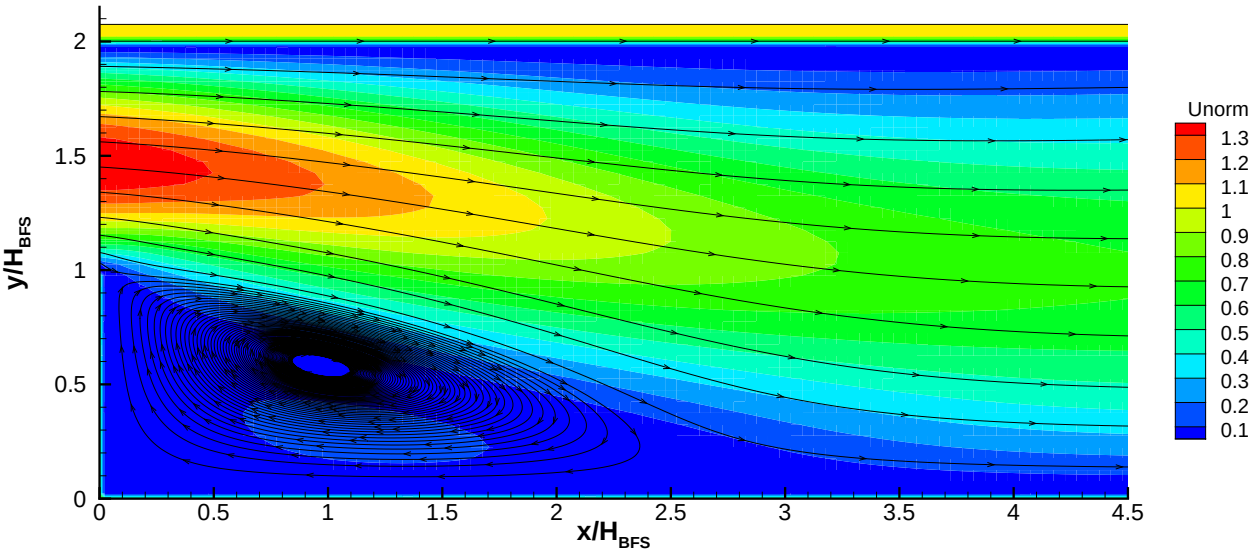



**Figure 4.23.** Streamlines earned with the TRT collision model

The dimensionless velocity profile (velocity divided by the inlet velocity) was plotted for the serial and for the parallel codes too. Figures 4.24 and 4.25 show the dimensionless velocity plots at different cross sections downstream of the step. The former shows the serial, while the latter represents the parallel code given results. The figures show all the three collision models, which resulted in very similar results. One can see at $x/H_{BFS} = 0$ [−] that the velocity profile was slightly shifted towards the positive $y$ values. This means that closer to the upper wall, the velocity profile was better resolved, while closer to the lower wall, the velocity was underestimated by all the three collision

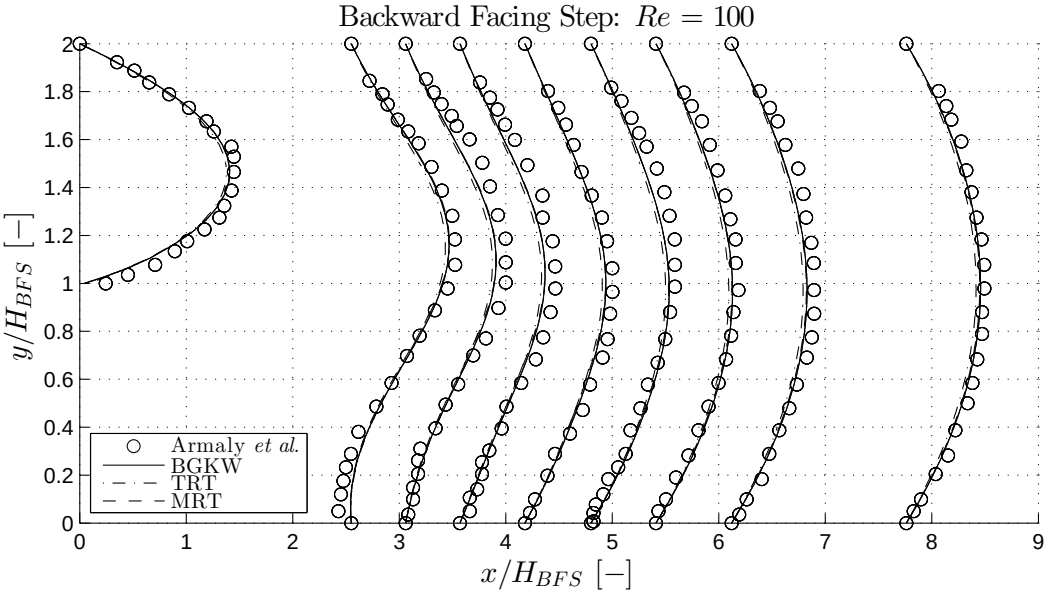



**Figure 4.24.** The dimensionless velocity profiles in different sections of the domain: earned with the *serial* code (origin is located in the lower left corner after the step)

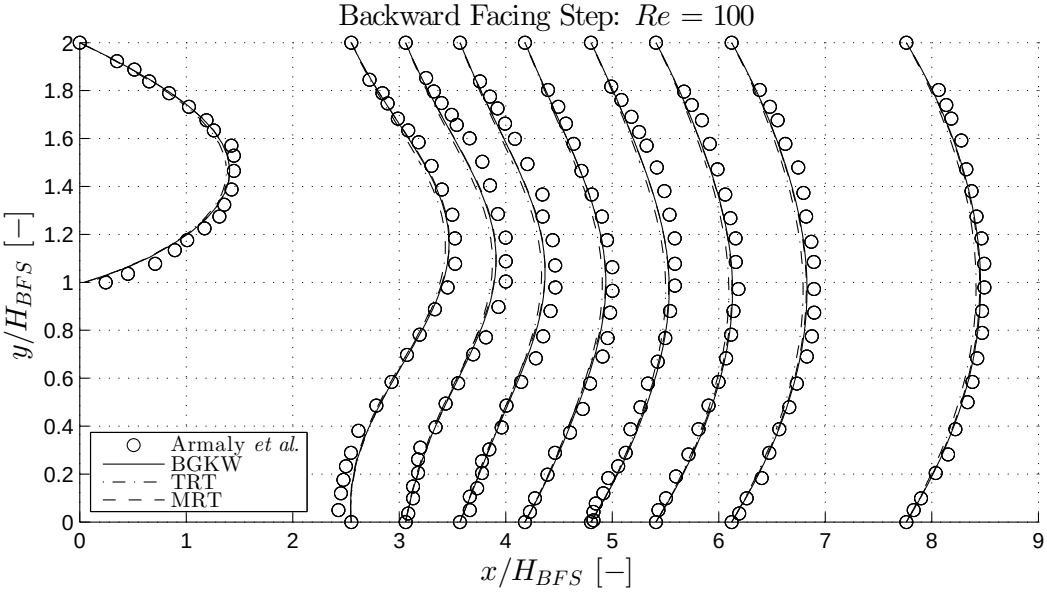



**Figure 4.25.** The dimensionless velocity profiles in different sections of the domain: earned with the *parallel* code (origin is located in the lower left corner after the step)

models. We can see at $x/H_{BFS} = 2.5$ [−] that the recirculation zone was not resolved accurately, i.e. the solvers underestimated the reattachment length. According to the measurements (drawn by circles), at this cross section we shall find a region, where negative velocity occur. The solvers overestimated the velocity in this region, i.e. only a slight negative velocity occurred. At the other cross sections the models generally

*underestimated the velocity magnitude.* As we move downstream, the results met the experiments better. Finally, one wish to note that from qualitative point of view, the velocity profiles met the experimental results.

One can state that, also in this case, both the serial and parallel codes were able to resolve the physics, and no remarkable difference was found between the results.

### 4.4.3 Sudden Expansion

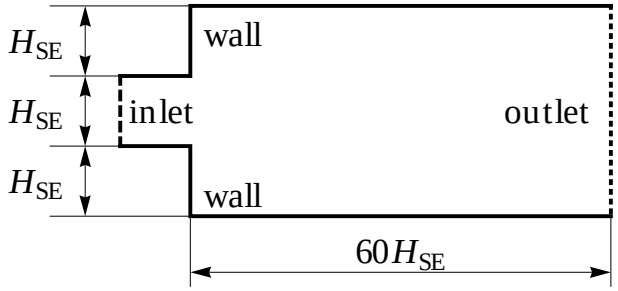



**Figure 4.26.** Geometry of the sudden expansion validation case

The third validation case was the sudden expansion. Similarly to the backward facing step, in this case we had experimental results available. Fearn *et al.* [50] measured such flow. The geometry of the domain and the boundary conditions were shown in Figure 4.26. We can see that the flow enters from a channel to a region, which has a height of $3H_{SE}$. This domain was more complex than for the backward facing step, since the flow finds two sharp edges at the expansion. The origin was located in the centre of the throat where the expansion occurs. Fearn *et al.* [50] defined the Reynolds number as

$$Re = \frac{u_{SE}{}^{N_{SE}/2}}{\nu_L}, \tag{4.4}$$

where $u_{SE}$ was the maximum velocity occurring at the inlet (3/2 of the mean entering velocity) and ${}^{N_{SE}}/_2$ was half of the number of nodes at the inlet. $\nu_L$ was the lattice viscosity. Two Reynolds numbers were considered in this validation case. The first was a lower one: $Re = 25$. The higher one was $Re = 80$, where despite of the geometrical symmetry, the flow looses its symmetry.

First, the velocity contours of the lower Reynolds number case was plotted in Figure 4.27. We can see in the figure that the flow remains symmetrical. After the expansion a free jet forms, which is surrounded by two corner vortices. These vortices are the same in size. The figure shows the normalized velocity contours along with the streamlines. The shown results were earned with the MRT collision model. The resolved vortices and the symmetry of the free jet met the expectations.

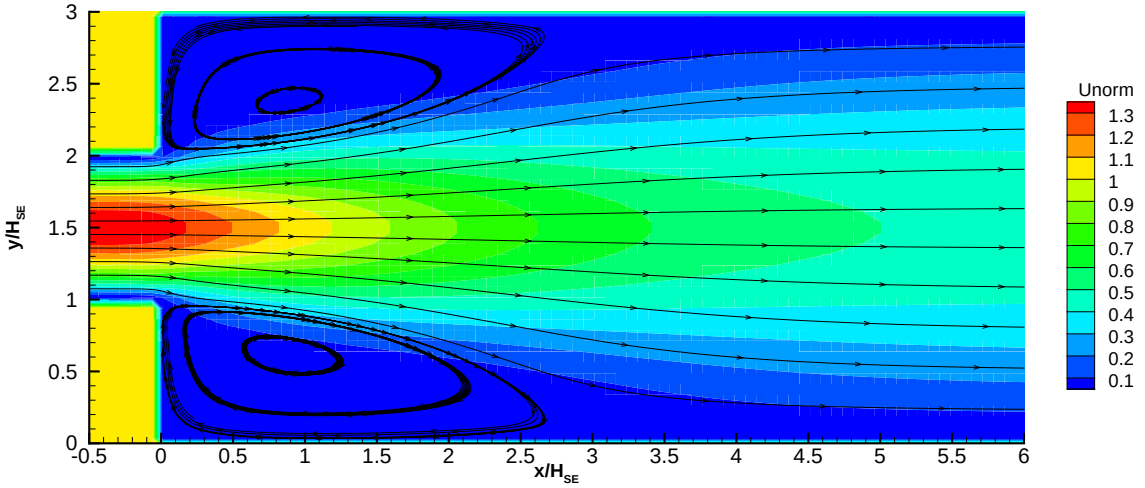



**Figure 4.27.** Streamlines for the $Re = 25$ case (MRT collision model)

In the second case, the Reynolds number was 80. Figure 4.28 shows the normalized velocity contours and the streamlines of this flow (MRT collision model). We can see that the flow lost its stability, and three vortices formed in the domain. Note that during the simulations the stable solution showed up first, and after a certain number of iterations, the free jet bent. For this reason, this validation case required the highest number of iterations. This showed well the robustness of the method, i.e. that the flow remained stable for a high number of iterations until, presumably, enough numerical error occurred which drove the flow toward the bent solution.

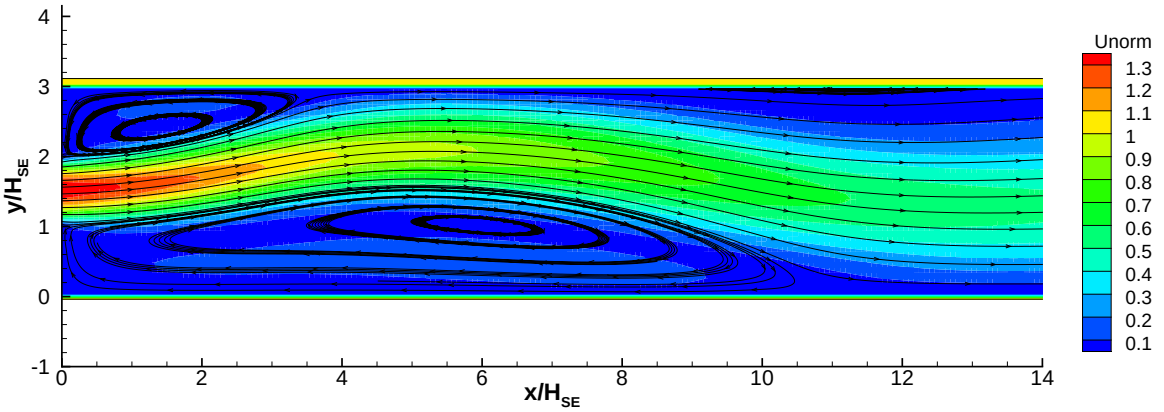



**Figure 4.28.** Streamlines for the $Re = 80$ case (MRT collision model)

Fearn *et al.* showed [50] that in this case three vortices show up in the flow. Two of them are in the corners, a smaller and a bigger one. The third one happens after the straightening of the free jet on the opposite side to the large vortex. All of these

vortices are visualized in Figure 4.28 with the shown streamlines. Based on this, the solver was capable for resolving the flow features.

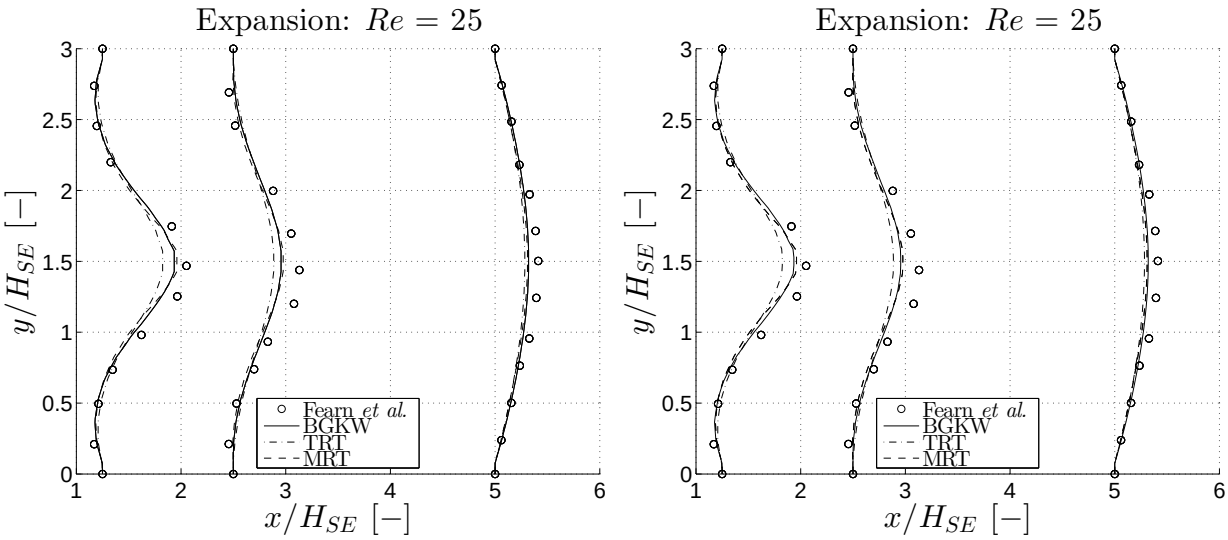



**Figure 4.29.** Dimensionless velocity profiles for the $Re = 25$ case, left: serial results, right: parallel results

In the followings, we compare the velocities to the available experimental data. Fearn *et al.* measured [50] the velocity magnitude downstream of the sudden expansion at different cross sections. These results were compared to the results earned with the serial and the parallel code. Figure 4.29 shows the dimensionless velocity for the lower Reynolds number flow, whilst Figure 4.30 shows the same for the $Re = 80$ case. In both figures, one finds the serial results on the left, and the parallel results on the right hand side. Note that there were no major differences between the results given by the two codes. Based on this, we can state (as formerly too) that the two codes were resolving the same physics.

For the lower Reynolds number ($Re = 25$), we can see that, similarly to the channel flow, all the three collision models *underestimated the velocity magnitude*. In addition to this, the velocity peak was also smoothed by the solver, i.e. the parabola-like profiles were resolved in a wider region. The higher Reynolds number flow ($Re = 80$, Figure 4.30) was better resolved by all the collision models. All the three models were very close to each other in this case and the results were very close to the experimental data. There was no significant difference between the two codes in this case either.

The reattachment length was measured for both Reynolds numbers and for all the collision models (based on streamlines). Table 4.5 and 4.6 shows the results. The former table lists the $Re = 25$ case and the latter one shows the $Re = 80$ case. The tables show similar trends as before. The reattachment lengths were *underestimated*, whilst the TRT model showed the weakest result in this case too.

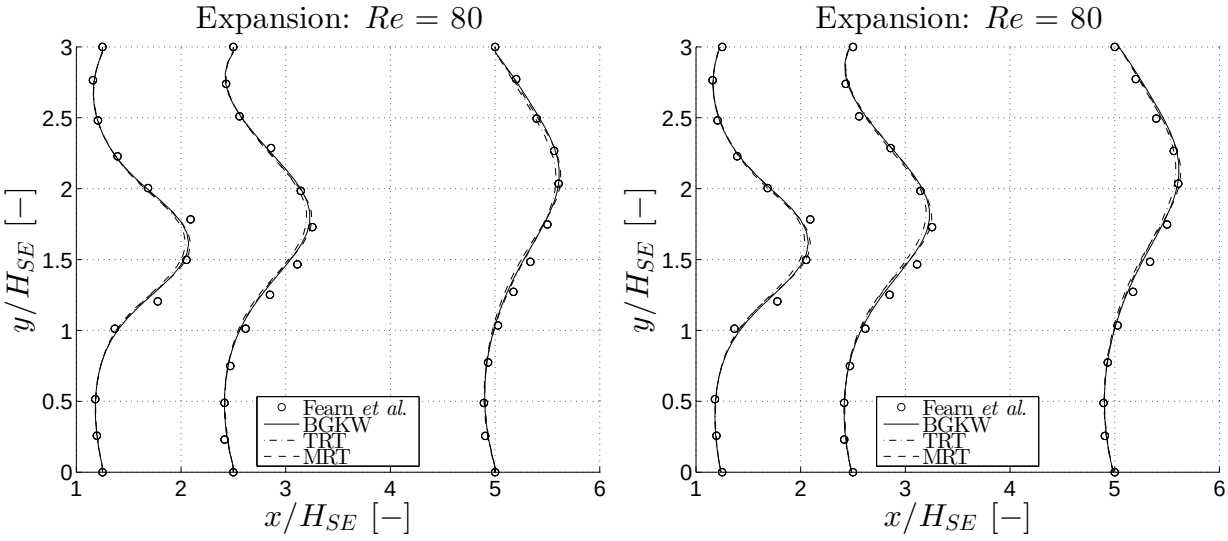



**Figure 4.30.** Dimensionless velocity profiles for the $Re = 80$ case, left: serial results, right: parallel results

| **Collision model** | **Simulation** | **Measurement** | **Error** [%] |
|---|---|---|---|
| BGKW | 3.35 | 3.435 | -2.47 |
| TRT | 3.28 | 3.435 | -4.50 |
| MRT | 3.30 | 3.435 | -3.90 |

**Table 4.5.** Reattachment length of the smaller and bigger vortices for sudden expansion: $Re = 25$

| **Collision model** | **Vortex** | **Simulation** | **Measurement** | **Error** [%] |
|---|---|---|---|---|
| BGKW | Bigger | 10.65 | 11.76 | -9.44 |
| | Smaller | 3.45 | 3.68 | -6.25 |
| TRT | Bigger | 10.57 | 11.76 | -10.0 |
| | Smaller | 3.64 | 3.68 | -1.09 |
| MRT | Bigger | 10.60 | 11.76 | -9.86 |
| | Smaller | 3.56 | 3.68 | -3.26 |

**Table 4.6.** Reattachment length of the smaller and bigger vortices for sudden expansion ($Re = 80$)

### 4.4.4 Flow Around a Cylinder

The fourth validation case was relying on the well known von Kármán[1] [54, 55, 56] vortex street. To achieve such flow phenomenon, one needs to introduce e.g. a

[1]One of the most famous Hungarian scientist. He was mechanical engineer, physicist and mathematician. In the age of *six*, he was able to calculate the multiplication of two four-four digit numbers using mental arithmetic. The solution of the vortices shedding from for example a cylinder helped him to explain the reason for the collapse of the Tacoma bridge.

cylinder into the flow. Theodore von Kármán derived the analytical solution for such complex flow. Figure 4.31 shows the geometry of the validation case. Note that this flow phenomenon requires time-dependent flow resolution. Until this point the resolved flows were steady (transient process was experienced in the sudden expansion case ).

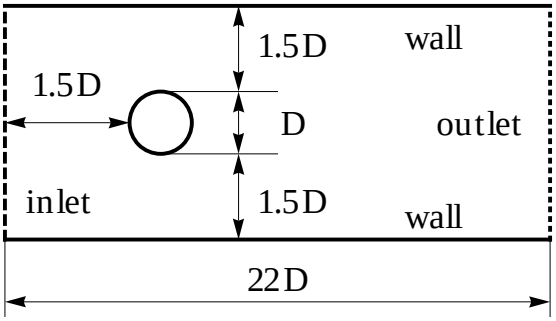



**Figure 4.31.** Flow around a cylinder: geometry and boundary conditions

The Reynolds number of such flow is defined based on the diameter ($D$) of the cylinder. In our case this reads as

$$Re = \frac{N_D u_D}{\nu_L}, \tag{4.5}$$

where $N_D$ is the number of nodes along the cylinder and $u_D$ is the upstream mean velocity. $\nu_L$ is the lattice viscosity. Vortices supposed to shed from a Reynolds number of around 50. For this reason, the validation was made for $Re = 100$.

Figure 4.32 shows the flow field around the cylinder after a few hundred of iterations. The left sub-figure shows us the contours of the normalized velocity magnitude (divided by $u_D$) and the right sub-figure visualises the vorticity field. We can see that at this initial state of the simulation, the flow remained symmetrical and stable. Later on, the stability of the flow downstream of the cylinder broke and von Kármán vortices started to shed.

Figure 4.33 shows the normalized velocity (divided by $u_D$) and the streamlines downstream the cylinder after a few thousand of iterations. We can see that a vortex is about to shed behind the cylinder. The streamlines show the path of the von Kármán vortex street. The figure shows that the solver was capable of resolving the vortices. From the normalized $x$ and $y$ velocities, the vorticity magnitude was calculated. Figure 4.34 shows the vorticity contours downstream the flow. It can be seen that from above of the cylinder, a new vortex is forming. From below, a vortex has just been shed at the point when the iteration was stopped (and the results were plotted).

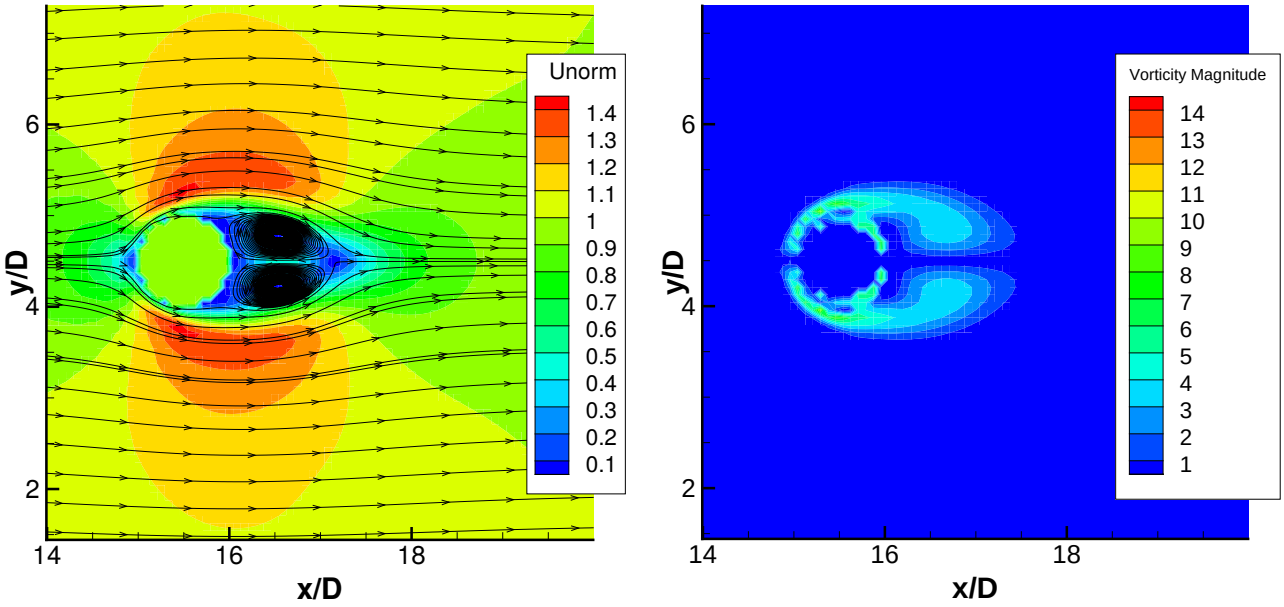



**Figure 4.32.** Velocity (left) and vorticity (right) contours of flow around a cylinder after a few hundred of iterations

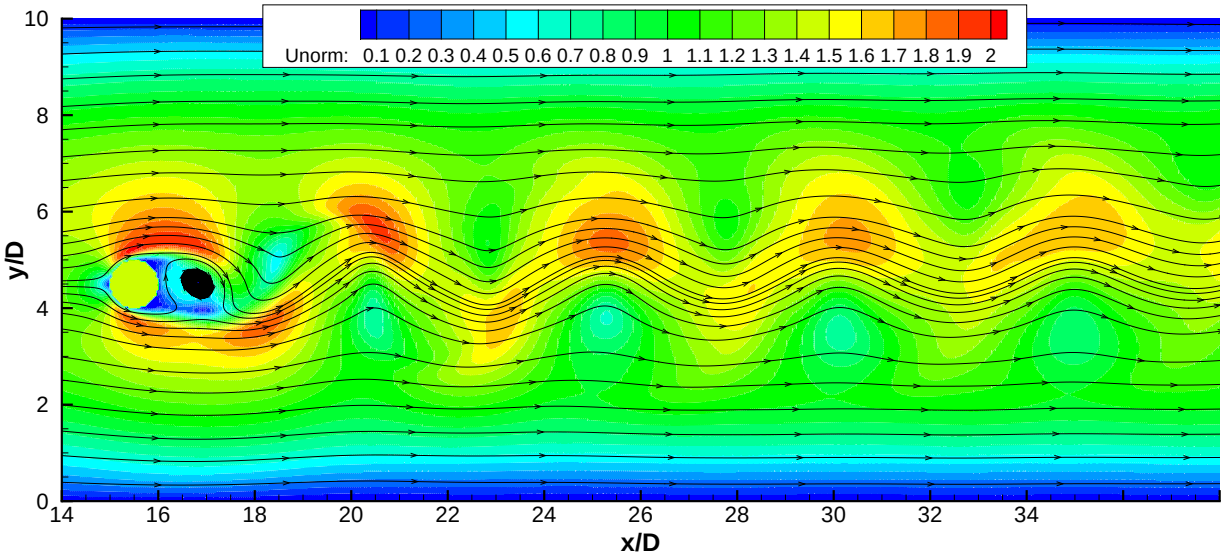



**Figure 4.33.** Velocity contours and streamlines of flow around a cylinder

One finds the derivation of the pressure distribution around a cylinder [20] in such books, which discuss the basics of fluid dynamics. The pressure coefficient on the circumference on the cylinder assuming inviscid laminar flow (no separation) reads as

$$C_P(\theta) = 1 - 4\sin^2(\theta), \tag{4.6}$$

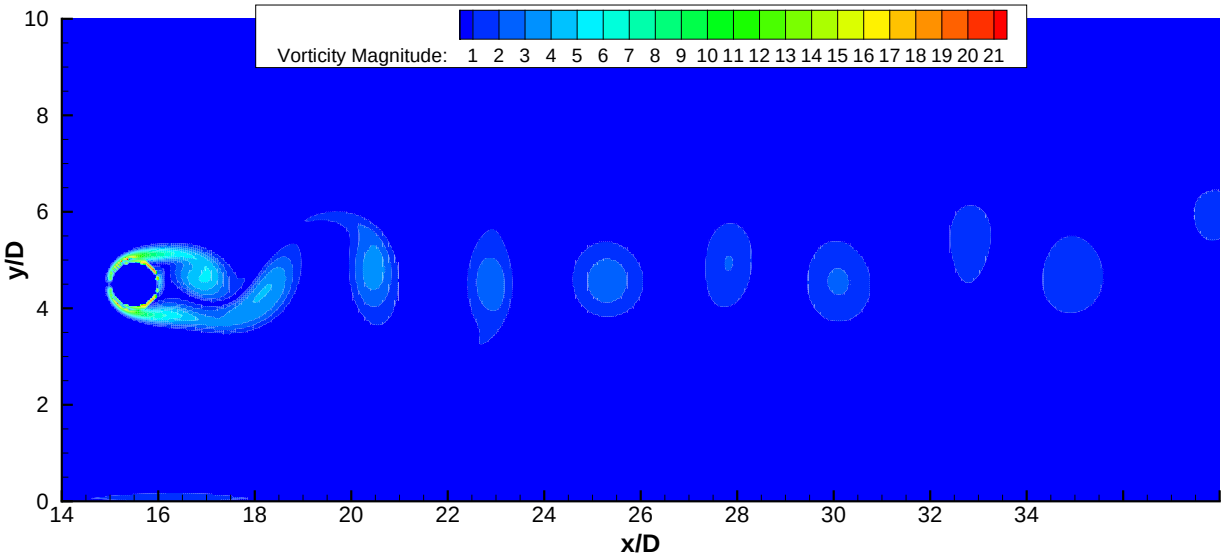



**Figure 4.34.** Vorticity contours of flow around a cylinder

where $\theta$ is the angle measured clockwise from the direction of $-\mathbf{i}$ (negative $x$ unit vector). This means that the front of the cylinder is $\theta = 0$ and the back of it is $\theta = \pi$. Figure 4.35 shows the analytical pressure coefficient with thick line, and it also shows the results of the three collision models. We can see that on the upper and the lower side (at $\theta = \pi/2$ and at $\theta = 3\pi/2$) the flow separates. At the same point wiggles are experienced. The suction is slightly shifted toward the back of the cylinder. Note that on the front of the cylinder one finds an approximately constant pressure ($\theta \in [0, \approx \pi/4]$ & $\theta \in [7\pi/4, 2\pi]$). We assume that this was caused by the step-wise resolution of the geometry (see Figure 4.3).

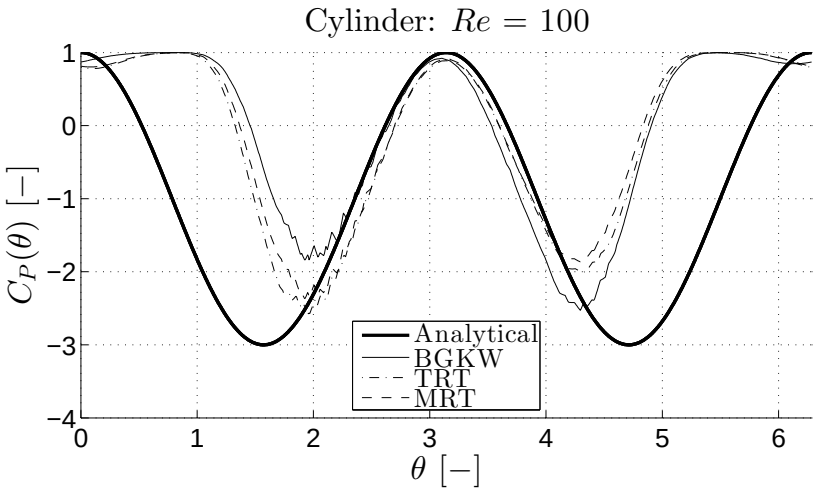



**Figure 4.35.** Pressure coefficient around the cylinder

### 4.4.5 Lid Driven Cavity

The last validation case was the lid driven cavity. In this case, the results were compared to the simulations of Ghia *et al.* [6] and Perumal & Dass [51]. Prasad and Koseff published measurement results for the lid driven cavity flow [57]. The geometry can be seen in Figure 4.36. The figure shows a cavity with aspect ratio of one, which is closed by a moving lid on the top. Other boundaries were set to be walls. The Reynolds number was defined as

$$Re = \frac{N_{Lid} u_{Lid}}{\nu_L}, \tag{4.7}$$

where $N_{Lid}$ was the number of cells along the $x$ (or $y$) direction and $u_{Lid}$ was the speed of the moving lid on the top. The initial density was set to be 5 [−] and the viscosity was $\nu_L = 0.01$ [−] in this case as well. Three different Reynolds numbers were simulated: $Re = 100$, $Re = 1000$ and $Re = 3200$. The used meshes and their properties can be found in Table 4.7.

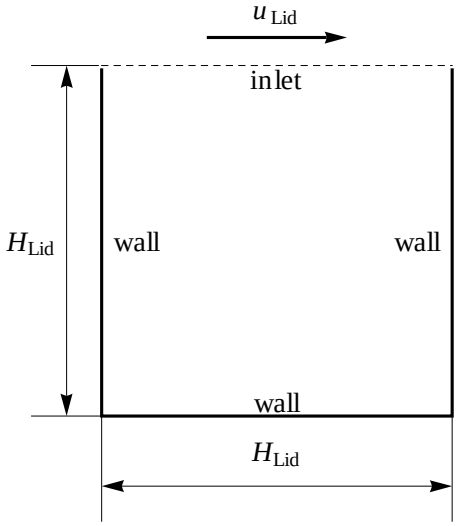



**Figure 4.36.** Geometry and boundary conditions of the lid driven cavity flow

| | **Coarse** | **Medium** | **Fine** |
|---|---|---|---|
| Grid spacing | 0.02 | 0.01 | 0.005 |
| Lid velocity | 0.2 | 0.1 | 0.05 |
| $N_x = N_y = N_{Lid}$ | 50 | 100 | 200 |

**Table 4.7.** The different meshes for the lid driven cavity flow

As formerly, we investigate the streamlines first. Figures 4.37, 4.38 and 4.39 show the streamlines plotted on the contours of the normalized (dividing by $u_{Lid}$) velocity

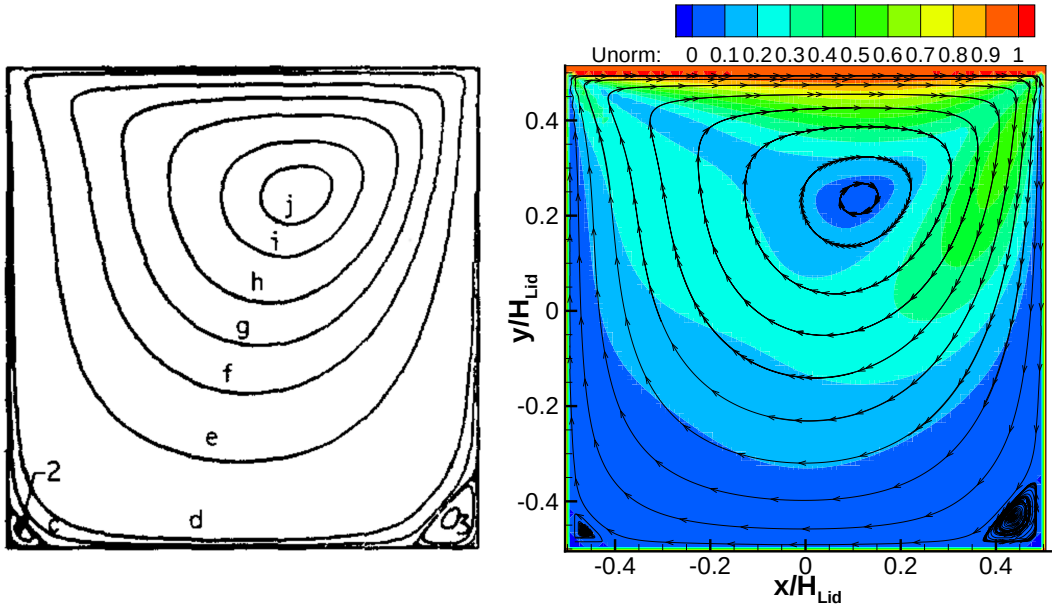



**Figure 4.37.** Streamlines of Ghia *et al.* [6] (left) compared to the simulated streamlines and normalised velocity contours (right) for the $Re = 100$ case

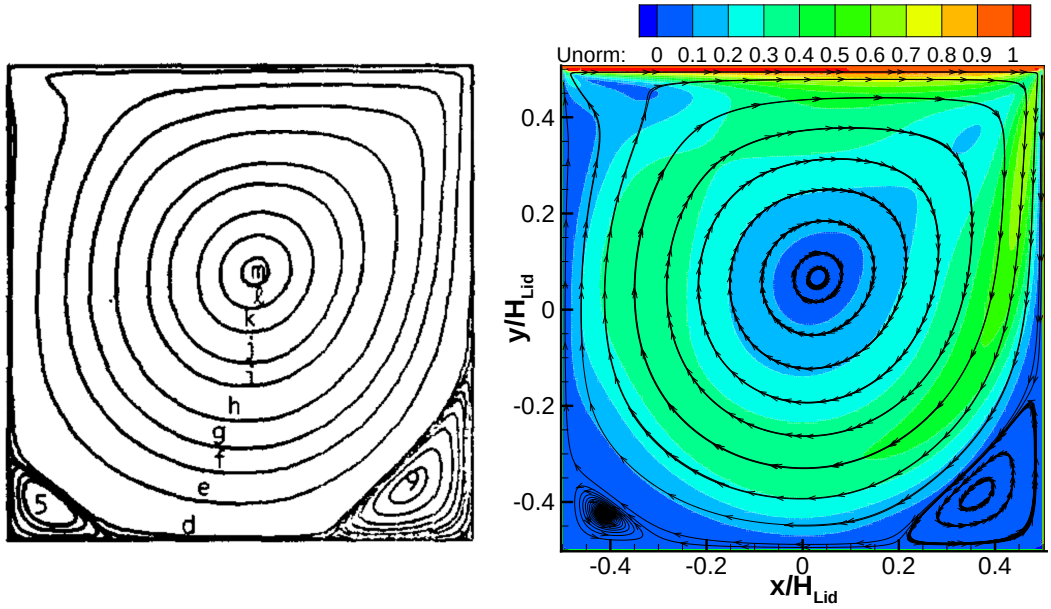



**Figure 4.38.** Streamlines of Ghia *et al.* [6] (left) compared to the simulated streamlines and normalised velocity contours (right) for the $Re = 1000$ case

magnitude for $Re = 100,\ 1000$ and 3200 respectively (BGKW collision model). In the first case, we find two smaller vortices in the domain in the lower corners. The left was a smaller vortex, while the right was a bigger one. The main vortex can be seen in the middle, which was slightly shifted up- and rightwards in the domain. We can see that

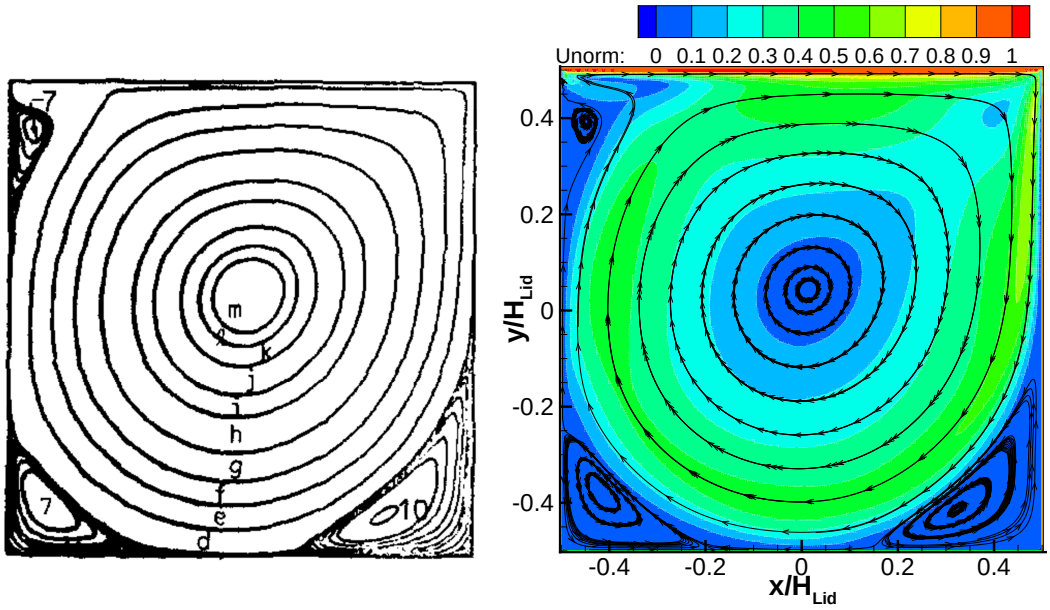



**Figure 4.39.** Streamlines of Ghia *et al.* [6] (left) compared to the simulated streamlines and normalised velocity contours (right) for the $Re = 3200$ case

compared to the figure of Ghia *et al.* [6], the flow structure was well resolved.

For the $Re = 1000$ case, the same statements can be made: two smaller vortices formed in the bottom corners. The left vortex was the smaller, while the right was the bigger one. The main vortex was almost in the middle of the domain. In terms of size and locations, the flow features were well resolved.

In the last case ($Re = 3200$), we can see that a new vortex was formed in the upper west corner, while the bottom left vortex became slightly bigger. The main vortex remained in the centre of the domain.

Figure 4.40 shows the dimensionless velocity magnitudes (velocity magnitude divided by $u_{Lid}$) along the horizontal and vertical centre of the domain for $Re = 100$. The left sub-figure shows the serial results, whilst the right sub-figure shows the parallel results. We can see that the three collision models gave quite similar results. The reason for the similarity (difference being within the thickness of the plotted line) was assumed to be because of the low Reynolds number. This means that due to the very low lid velocity, the magnitude of the velocities were small, and as we can see in the figure, the velocity profile was smooth. This validation case was the most stable one, since no outlet is present in the domain. The collision models also accepted low number of cells, while in the former cases some of them (MRT, TRT) was not able to run on very coarse meshes. The serial and the parallel codes gave similar results.

For $Re = 1000$, the same plots were created. Figure 4.41 shows the dimensionless velocities along the centre lines of the domain. The left sub-figure represents the results

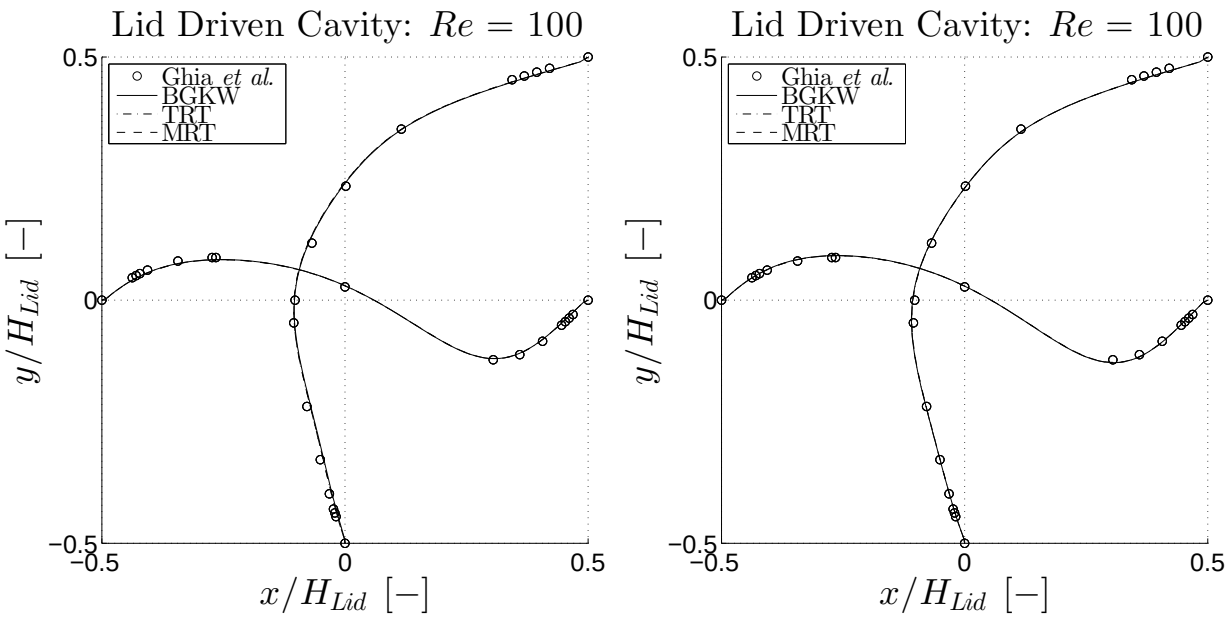



**Figure 4.40.** Dimensionless velocity profiles of $Re = 100$ case: serial results (left) and parallel results (right)

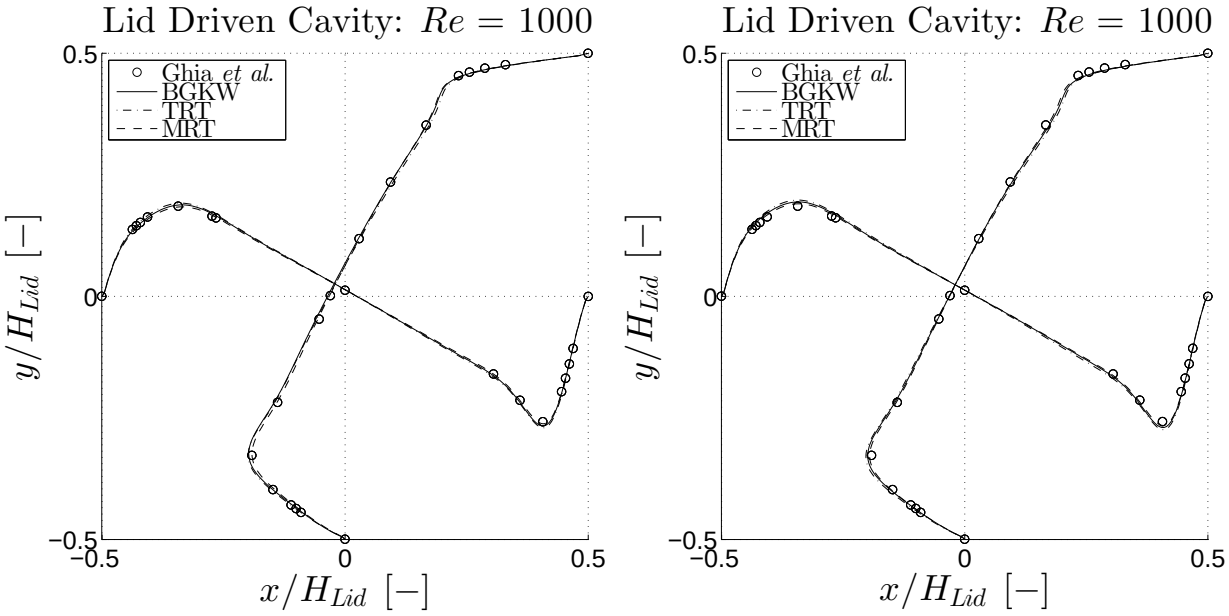



**Figure 4.41.** Dimensionless velocity profiles of $Re = 1000$ case: serial results (left) and parallel results (right)

of the serial code, while the right one shows us the results of the parallel code. As formerly, we can see that only slight differences occur in the two cases. The three collision models were capable of resolving such flow. At some points, the solver seemed to slightly overshoot the velocity magnitude. We can see that quite good agreement was found compared to the results of Ghia *et al.* [6].

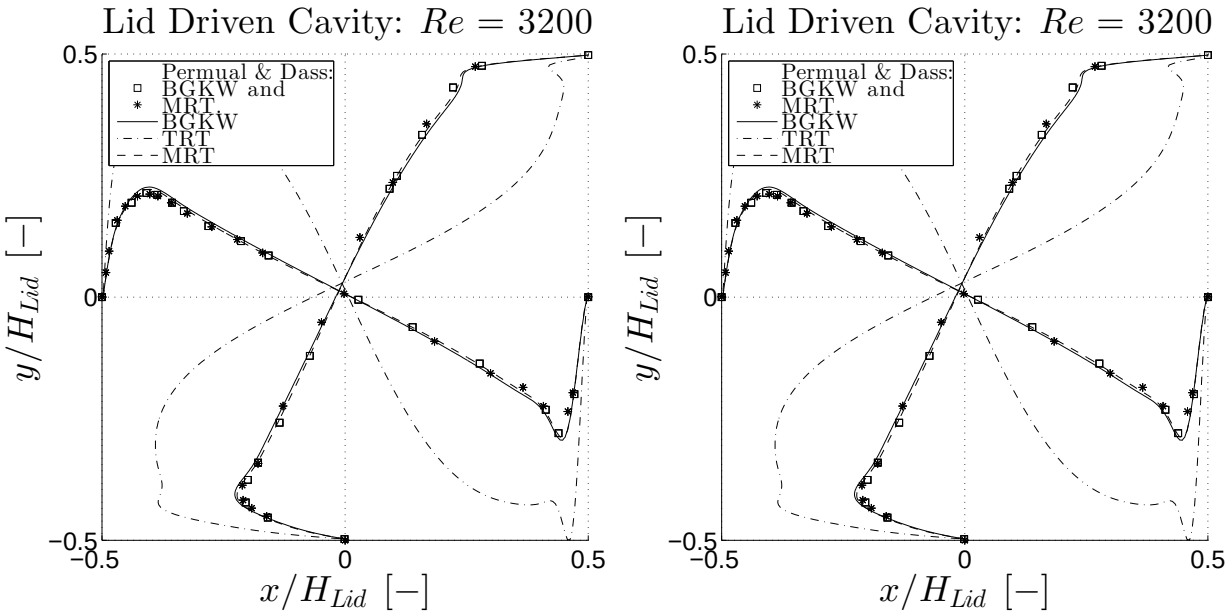



**Figure 4.42.** Dimensionless velocity profiles of $Re = 3200$ case: serial results (left) and parallel results (right)

The velocity plot of the $Re = 3200$ case shows slightly different results compared to the former results. Figure 4.42 compares the serial (left) and the parallel (right) results to the results of Permual & Dass [51], from where BGKW and MRT data was available. By taking a look at the figure, we can see that the TRT model gave unacceptable results for this case, whilst the MRT and BGKW results were still reliable. He & Luo [58] reported that the method has a practical limit for the lattice speed, which is defined in terms of lattice Mach number. This dimensionless value is defined as $Ma = u_L/c_s$, where $u_L$ is the maximum lattice speed occurring in the domain and $c_s = 1/\sqrt{3}$ is the lattice sonic speed. The rule of thumb in the community of the lattice Boltzmann method researchers [10, 12, 58] is to set $Ma = \sqrt{3}u_L < 0.15$. Note that in this case ($Re = 3200$) this speed was slightly exceeded. We dedicate the unacceptable results of the TRT model to this. The serial (left) and the parallel (right) codes gave the same results in this case as well.

Finally, Figure 4.43 shows the mesh convergence study of the flow. Three meshes, listed in Table 4.7, were used to perform the mesh convergence study. The figure shows the results earned with MRT collision model for $Re = 1000$. We can see that as the mesh became smoother, the velocity peaks slightly decreased. Altaugh, moving from medium to fine mesh, the change was almost within the thickness of the plotted line.

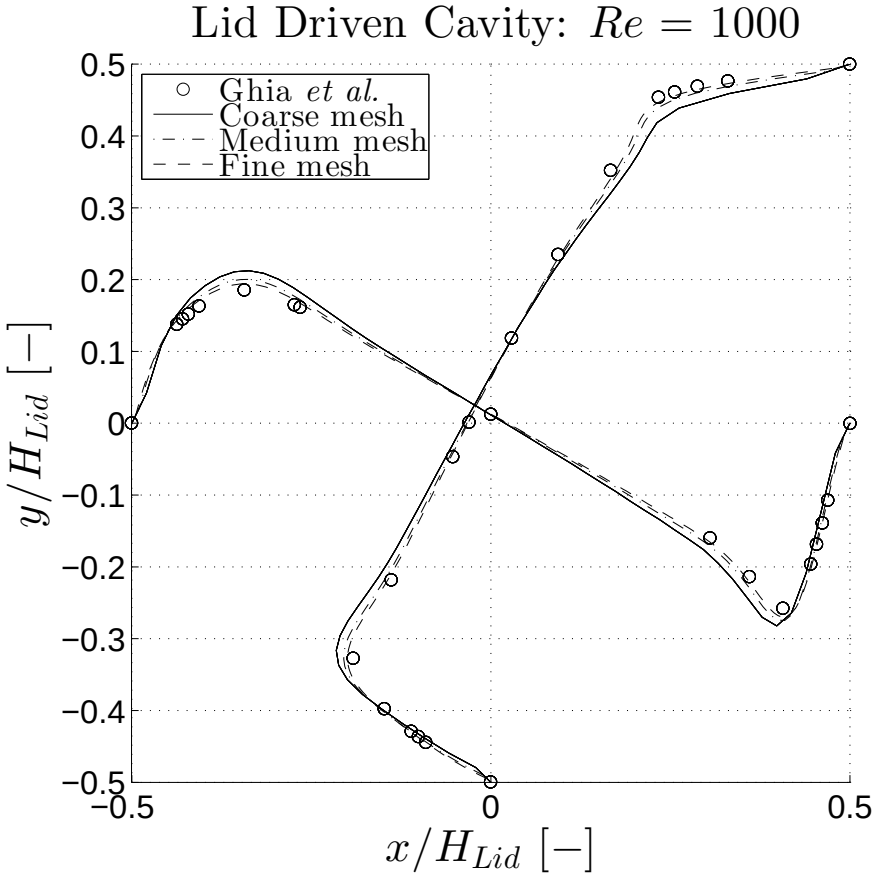



**Figure 4.43.** Lid mesh convergence study

# 5 Conclusions

This thesis discussed the parallelisation of existing in-house lattice Boltzmann solvers [2, 9]. From two C++ based in-house codes, a new C code was born. It inherits advantages from both previous codes, while offering further simplicity and readability. This transformation from C++ to C was a non-trivial step, as many of the advantageous features of C++ for this code are not available in C. The user finds new features in the C code, which was developed with the cooperation of Tamás Józsa [46]. Up to this point, we can say that since C is a lower-level language than C++, it offers better speed while requiring more conceptual effort.

The purpose of this study was to parallelise the new C code, which was the next step. Unified Parallel C (UPC) [3, 13], which is a Partitioned Global Address Space (PGAS) [19] based language, was used to develop the parallel solver. It offers the user straightforward and easy-to-use new techniques compared to the Message Passing Interface (MPI) approach. These were found to perform poorly. As a next step, the conventional MPI-like data layout was developed. Nota bene: Even if the idea behind the parallelisation is the same as it would be in MPI, using UPC still requires less conceptual effort, offers better readability and requires fewer lines to achieve the same goals [23]. It was reported by several papers that the gained speed-up may be better than in the conventional approaches like MPI [26, 27, 29]. This was experienced in the new parallel code as well. The gained speed-up exceeded the given traditional limitations. The developed code was faster than the limit meant by the ideal speed-up. As a conclusion of this work, we can state that one wish to avoid the easier and faster parallelisation approach (shared variables). MPI, which is a local variable based programming model, became dominant for this reason. One can gain such speed in UPC as well (via local variables), which the programmer should aim to reach. From this point, we can assume that in the future of parallel programming, the idea of the MPI approach is going to be conserved. At the same time, the compiler tends to take more responsibility from the user (overlapping processes, data handling, etc.). This also increases the portability of any code. All of these features will help non-expert programmers such as engineers. We can assume that for this reason the PGAS programming languages are good tools for implementing engineering based problems.

The validation of both the new serial and the better parallel code was done. The poorly performing shared variable based code was not considered for validation. The five flow simulation cases showed that both the new codes are capable of solving such problems. The channel flow is a basic laminar validation problem, where we found that the codes gave underestimated velocities. The backward facing step and the sudden expansion validation cases showed that the code is capable of resolving separations, of

which the length was also underestimated. The unsteady approach of the method was tested on the von Kármán vortex street. At this point, we found that the solver gave qualitatively good results. Finally, the lid driven cavity was the most stable validation case, which also gave good agreement with the measurements. We assume that the underestimation of the velocities were a result of the dissipative property of the method.

The novelty introduced by this thesis relies on the non-conventional parallelisation approach. As a continuation of this work, the next reasonable step would be to broaden the possibilities of the solver. The most relevant step would be to extend it to three dimensions, which would need a new mesher. This is not necessary to develop in C. The author suggest to use environments where a graphical user interface is offered, such as Python or Matlab. Other opportunities lie in the implementation of multi-speed models. Such approaches are capable of resolving vortical flows better than the actual solver (see Lycett-Brown *et al.* [59]). Finally, we suggest the implementation of additional boundary conditions to the code. This step should be done with great care as the method is very sensitive to the boundary conditions.

The author hopes that this thesis not only provides a valuable addition to the former codes, but it also opens the possibilities of further evolving it. Since the coding was done bearing readability and the simplicity in mind, it forms a good basis for future studies.

# References

[1] Top500. Top500 Poster, June 2014. Technical report. URL http://www.top500.org/. Accessed 16/08/2014.

[2] T. R. Teschner. Development of a Two Dimensional Fluid Solver Based on the Lattice Boltzmann Method. Master's thesis, Cranfield University, 2013.

[3] S. Chauwvin, P. Saha, F. Cantonnet, S. Annareddy, T. El-Ghazawi. *UPC Manual*. The George Washington University, Washington, DC. Version 1.2.

[4] O. Serres, A. Kayi, A. Anbar, T. El-Ghazawi. Hardware Support for Address Mapping in PGAS Languages; a UPC Case Study. *CoRR*, abs/1309.2328, 2013.

[5] P. Husbands, C. Iancu, K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 63–73, 2003.

[6] U. Ghia, K. N. Ghia, and C. T. Shin. High-re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method. *Journal of Computational Physics, vol. 48, no. 3, pp. 387–411*, 1982.

[7] P. L. Bhatnagar, E. P. Gross, and M. Krook. A Model for Collision Processes in Gases. I.Small Amplitude Processes in Charged and Neutral One-component Systems. *Physical Review vol. 94, no. 3, p. 511–525*, 1954.

[8] P. Welander. On the Temperature Jump in a Rarefied Gas. *Arkiv for Fysik, vol. 7, no. 6, pp. 507–533*, 1954.

[9] G. Abbruzzese. Development of a 2D Lattice Boltzmann Code. Master's thesis, Cranfield University, 2013.

[10] S. Succi. *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*. Oxford, 2001.

[11] E. Fares. Unsteady Flow Simulation of the Ahmed Reference Body Using a Lattice Boltzmann Approach. *Comput. Fluids*, 35(8-9):940–950, 2006.

[12] A. A. Mohamed. *Lattice Boltzmann Method: Fundamentals and Engineering Applications With Computer Codes*. Springer, London, 2011.

[13] T. El-Ghazawi, W. Carlson, T. Sterling, K. Yelick. *UPC: Distributed Shared Memory Programming*. Wiley, 2005.

[14] J. von Neumann. *The Computer and the Brain*. Yale University Press, 2000.

[15] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics Magazine. p. 4.*, 1965.

[16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. 2012. URL http://www.mpi-forum.org/. Accessed 16/08/2014.

[17] OpenMP Archiecture Review Board. *OpenMP Application Program Interface*. openmp.org, 2011. URL `http://www.openmp.org/`. Accessed 16/08/2014.

[18] nVidia. *CUDA Toolkit Documentation v6.0*. nVidia, 2014.

[19] Partitioned Global Address Space. URL `http://www.pgas.org/`. Accessed 16/08/2014.

[20] P. K. Kundu, I. M. Cohen. *Fluid Mechanics*. Elsevier, 2010.

[21] D. C. Wilcox. *Turbulence Modeling for CFD*. DCW Industries, Inc., 2006.

[22] I. Ginzburg. Variably Saturated Flow Described with the Anisotropic Lattice Boltzmann Methods. *Computers and Fluids vol. 35, no. 8-9, pp. 831–848*, 2006.

[23] F. Cantonnet, Y. Yao, M. Zahran, T. El-Ghazawi. Productivity Analysis of the UPC Language. pages 254–, April 2004.

[24] ISO C Language Standard.

[25] T. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed. Benchmarking Parallel Compilers: A UPC Case Study. *Future Generation Computer Systems*, 22(7):764 – 775, 2006. ISSN 0167-739X.

[26] D. A. Mallón, A. Gómez, J. C. Mouriño, G. L. Taboada, C. Teijeiro, J. Touriño, B.B. Fraguela, R. Doallo, B. Wibecan. UPC Performance Evaluation on a Multicore System. pages 9:1–9:7, 2009.

[27] G.L. Taboada, C. Teijeiro, J. Tourio, B.B. Fraguela, R. Doallo, J. C. Mourino, D. A. Mallon. Performance Evaluation of Unified Parallel C Collective Communications. *High Performance Computing and Communications, HPCC '09. 11th IEEE International Conference on High Performance Computing and Communications , vol., no., pp.69,78, 25-27*, 2009.

[28] URL `http://gasnet.lbl.gov/`. Accessed 19/08/2014.

[29] S. Markidis, G. Lapenta. Development and Performance Analysis of a UPC Particle-in-Cell Code. pages 10:1–10:9, 2010.

[30] A. A. Johnson. Unified Parallel C Within Computational Fluid Dynamics Applications on the Cray X1(E). *CUG Proceedings*, 2005.

[31] Cray Inc. *Cray C and C++ Reference Manual*. Cray Inc., 2012.

[32] F. Mandl. *Statistical Physics*. John Wiley & Sons, 2008.

[33] J.C. Maxwell. Illustrations of the Dynamical Theory of Gases. Part I. On the Motions and Collisions of Perfectly Elastic Spheres. *Philosophical Magazine*, 1860.

[34] J.C. Maxwell. Illustrations of the Dynamical Theory of Gases. Part II. On the Process of Diffusion of Two or More Kinds of Moving Particles Among One Another. *Philosophical Magazine*, 1860.

[35] Z. Chai, B. Shi, Z. Guo, F. Rong. Multiple-relaxation-time Lattice Boltzmann Model for Generalized Newtonian Fluid Flows . *Journal of Non-Newtonian Fluid Mechanics*, 166(5–6):332 – 342, 2011.

[36] R. Du, W. Liu. A New Multiple Relaxation Time Lattice Boltzmann Method for Natural Convection. *Journal of Scientific Computing*, 2013.

[37] K. N. Premnath, J. Abraham. Three-dimensional Multi-relaxation Time (MRT) Lattice-Boltzmann Models for Multiphase Flow. *Journal of Computational Physics*, 224(2):539 – 559, 2007. ISSN 0021-9991.

[38] R. Courant, K. Friedrichs, H. Lewy. On the Partial Difference Equations of Mathematical Physics. *AEC Research and Development Report, NYO-7689*, 1928.

[39] Y. B. Bao, J. Meskas. Lattice Boltzmann Method for Fluid Simulations. 2011.

[40] Q. Zou, X. He. On Pressure and Velocity Boundary Conditions for the Lattice Boltzmann BGK Model. *Physics of Fluids, vol. 9, no. 6, pp. 1591–1598*, 1997.

[41] S. H. Kim, H. Pitsch. On the Lattice Boltzmann Method for Multiphase Flows. *Center for Turbulence Research Annual Research Briefs*, 2009.

[42] X. He, S. Chen, R. Zhang. A Lattice Boltzmann Scheme for Incompressible Multiphase Flow and Its Application in Simulation of Rayleigh-Taylor Instability. *Journal of Computational Physics 152, 642–663*, 1999.

[43] M. Ashrafizaadeh and H. Bakhshaei. A Comparison of Non-Newtonian Models for Lattice Boltzmann Blood Flow Simulations . *Computers & Mathematics with Applications*, 58(5):1045 – 1054, 2009. doi: http://dx.doi.org/10.1016/j.camwa.2009.02.021. Mesoscopic Methods in Engineering and Science.

[44] Y. Liu. A Lattice Boltzmann Model For Blood Flows. *Applied Mathematical Modelling*, 36(7):2890 – 2899, 2012. doi: http://dx.doi.org/10.1016/j.apm.2011.09.076.

[45] Ch-H. Wang and J-R. Ho. A Lattice Boltzmann Approach for the Non-Newtonian Effect in the Blood Flow . *Computers & Mathematics with Applications*, 62(1):75 – 86, 2011. doi: http://dx.doi.org/10.1016/j.camwa.2011.04.051.

[46] T. I. Józsa. Parallelisation of Lattice Boltzmann Method Using Cuda Platform. Master's thesis, Cranfield University, 2014.

[47] B. W. Kernighan, D. M. Ritchie. *The C Programming Language*. Prentice Hall Software Series, 1988.

[48] J. Sanders, E. Kandrot. *CUDA by Example: An Introduction to General-purpose GPU Programming*. Addison-Wesley Professional, 2010.

[49] B. F. Armaly, F. Durst, J. C. F. Pereira, and B. Schönung. Experimental and Theoretical Investigation of Backward-facing Step Flow. *Journal of Fluid Mechanics vol. 127, pp. 473–496*, 1983.

[50] R. M. Fearn, T. Mullin, and K. A. Cliffe. Nonlinear Flow Phenomena in a Symmetric Sudden Expansion. *Journal of Fluid Mechanics, vol. 211, pp. 595–608*, 1989.

[51] D. A. Perumal and A. K. Dass. Application of Lattice Boltzmann Method for Incompressible Viscous Flows. *Applied Mathematical Modelling, vol. 37, no. 6, pp. 4075–4092*, 2013.

[52] F. Durst, S. Ray, B. Unsal, O. A. Bayoumi. The Development Lengths of Laminar Pipe and Channel Flows. *Journal of Fluids Engineering, vol. 127, no. 6, pp. 1154–1160*, 2005.

[53] J. W. Slater. Examining Spatial (Grid) Convergence. *NASA*, 2008. URL `http://www.grc.nasa.gov/WWW/wind/valid/tutorial/spatconv.html`. Accessed 17/08/2014.

[54] T. von Kármán. *Aerodynamics*. McGraw-Hill, 1963.

[55] T. von Kármán. On the Mechanism of the Drag a Moving Body Experiences in a Fluid . *Progress in Aerospace Sciences*, 59(0):13 – 15, 2013. doi: http://dx.doi.org/10.1016/j.paerosci.2013.03.004. Special issue: Theodore von Kármán.

[56] T. von Kármán. On the Mechanism of the Drag a Moving Body Experiences in a Fluid . *Progress in Aerospace Sciences*, 59(0):16 – 19, 2013. doi: http://dx.doi.org/10.1016/j.paerosci.2013.04.001. Special issue: Theodore von Kármán.

[57] A. K. Prasad and J. R. Koseff. Reynolds Number and End Wall Effects on a Lid Driven Cavity Flow. *Physics of Fluids, vol. 1, no. 2, pp. 208–219*, 1989.

[58] X. He, L-S. Luo. Lattice Boltzmann Model for the Incompressible Navier-Stokes Equation. *Journal of Statistical Physics, 88(3-4):927–944*, 1997.

[59] D. Lycett-Brown, I. Karlin, and K. H. Luo. Droplet Collision Simulation by a Multi-speed Lattice Boltzmann Method. *" Communications in Computational Physics, vol. 9, no. 5, pp. 1219–1234*, 2011.

# Appendix A Benchmark Codes

Serial benchmark code.

```
// To compile & to run: icc c_incr.c -o cIncr && ./cIncr

#include <stdio.h>        // fprintf
#include <math.h>         // Mathematic expressions
#include <time.h>         // Time measurement

#define NN 100000
#define MM 10000
#define BLS NN/4

int main (int argc, char* argv[])
{

  double* pLocal;
  double   Local[BLS];
  int i, j;
  int lA=0,lB=0;
  float t_pLocal        = 0.0;
  float t_Local         = 0.0;
  clock_t tInstant1, tInstant2;

  pLocal   = calloc(BLS,sizeof(double));

  // pointer to local incrementation
  tInstant1 = clock();
  for(j=0; j<MM; j++)
  {
    for(i=0; i<BLS; i++)
    {
      pLocal[i] = pLocal[i]+1;
      lA++;
    }
  }
  tInstant2 = clock();
  t_pLocal = (float)(tInstant2-tInstant1) / CLOCKS_PER_SEC;
  printf("pLocal took %f seconds; counter = %d\n", t_pLocal, lA);

  // static local incrementation
    tInstant1 = clock();
  for(j=0; j<MM; j++)
  {
    for(i=0; i<BLS; i++)
```

```
    {
      Local[i] = Local[i]+1;
      lB++;
    }
  }
  tInstant2 = clock();
  t_Local = (float)(tInstant2-tInstant1) / CLOCKS_PER_SEC;
  printf("Local took %f seconds; counter = %d\n", t_Local, lB);
  free(pLocal);
}
```

Unified Parallel C benchmark code.

```
// To compile & to run: upcc -T=4 upc_shared.c -o upcShared && upcrun
   upcShared

#include <stdio.h>       // fprintf
#include <math.h>        // Mathematic expressions
#include <time.h>        // Time measuring
#include <upc_relaxed.h> // Required for UPC extensions

#define NN 100000
#define MM 10000
#define BLS NN/THREADS

shared [BLS] double *pShared;
shared [BLS] double Shared[NN];

int main (int argc, char* argv[])
{
  if(MYTHREAD==0)
    printf("BLS = %d\n", BLS);

  int i, j;
  int length = NN;
  double* pLocal;
  double Local[BLS];
  int sA=0,sB=0,lA=0,lB=0;
  float t_pShared       = 0.0;
  float t_Shared        = 0.0;
  float t_pLocal        = 0.0;
  float t_Local         = 0.0;
  clock_t tInstant1, tInstant2;

  upc_barrier;

```

```
33  pShared  = (shared [BLS] double*)upc_all_alloc(THREADS,BLS*sizeof(
     double));
34  pLocal   = calloc(BLS,sizeof(double));
35
36  // pointer to shared incrementation
37  tInstant1 = clock();
38  for(j=0; j<MM; j++)
39  {
40    upc_forall(i=0; i<NN; i++; &pShared[i])
41    {
42      pShared[i] = pShared[i]+1;
43      sA++;
44    }
45  }
46  tInstant2 = clock();
47  t_pShared = (float)(tInstant2-tInstant1) / CLOCKS_PER_SEC;
48  printf("TH%d ::: pShared took %f seconds; counter = %d\n", MYTHREAD,
      t_pShared, sA);
49
50  // static shared incrementation
51  tInstant1 = clock();
52  for(j=0; j<MM; j++)
53  {
54    upc_forall(i=0; i<NN; i++; &Shared[i])
55    {
56      Shared[i] = Shared[i]+1;
57      sB++;
58    }
59  }
60  tInstant2 = clock();
61  t_Shared = (float)(tInstant2-tInstant1) / CLOCKS_PER_SEC;
62  printf("TH%d ::: Shared took %f seconds; counter = %d\n", MYTHREAD,
     t_Shared, sB);
63
64  // pointer to local incrementation
65  tInstant1 = clock();
66  for(j=0; j<MM; j++)
67  {
68    for(i=0; i<BLS; i++)
69    {
70      pLocal[i] = pLocal[i]+1;
71      lA++;
72    }
73  }
74  tInstant2 = clock();
75  t_pLocal = (float)(tInstant2-tInstant1) / CLOCKS_PER_SEC;
```

```
76    printf("TH%d ::: pLocal took %f seconds; counter = %d\n", MYTHREAD,
       t_pLocal, lA);
77
78    // static local incrementation
79    tInstant1 = clock();
80    for(j=0; j<MM; j++)
81    {
82      for(i=0; i<BLS; i++)
83      {
84        Local[i] = Local[i]+1;
85        lB++;
86      }
87    }
88    tInstant2 = clock();
89    t_Local = (float)(tInstant2-tInstant1) / CLOCKS_PER_SEC;
90    printf("TH%d ::: Local took %f seconds; counter = %d\n", MYTHREAD,
       t_Local, lB);
91
92    // free pointers
93    upc_free(pShared);
94    free(pLocal);
95
96  } // END OF MAIN
```