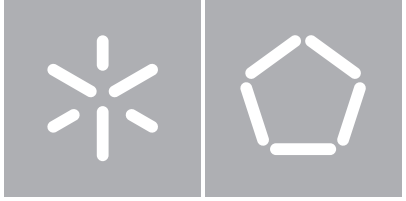
Universidade do Minho
Escola de Engenharia

David dos Santos Pereira

Numerical modeling of extrusion forming tools: improving its efficiency on heterogeneous parallel computers

Outubro de 2014



Universidade do Minho
Escola de Engenharia
Departamento de Informática

David dos Santos Pereira

Numerical modeling of extrusion forming tools: improving its efficiency on heterogeneous parallel computers

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Alberto José Proença
Nelson Daniel Ferreira Gonçalves

Outubro de 2014

Anexo 3

DECLARAÇÃO

Nome

David dos Santos Pereira

Endereço electrónico: pg22821@alunos.uminho.pt Telefone: 911530743 / _____

Número do Bilhete de Identidade: 13910400

Titulo dissertação ■/tese □

Numerical modeling of extrusion forming tools: improving its efficiency on heterogeneous parallel computers

Orientador(es):

Alberto José Proença

Nelson Daniel Ferreira Gonçalves Ano de conclusão: 2014

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Engenharia Informática

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

1. É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

2. É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE/TRABALHO (indicar, caso tal seja necessário, nº máximo de páginas, ilustrações, gráficos, etc.), APENAS PARA EFEITOS DE INVESTIGAÇÃO, , MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

3. DE ACORDO COM A LEGISLAÇÃO EM VIGOR, NÃO É PERMITIDA A REPRODUÇÃO DE QUALQUER PARTE DESTA TESE/TRABALHO

Universidade do Minho, 31 / 10 / 2014

Assinatura: David dos Santos Pereira

Agradecimentos

Ao meu orientador, Alberto Proença, que me acompanhou não só nesta dissertação bem como nestes dois anos de mestrado e por ter sempre exigido o máximo de mim. Agradeço todos os seus excelentes conselhos nos momentos mais difíceis e o seu acompanhamento sempre muito próximo. Ao meu co-orientador, Nelson Gonçalves, autor do código utilizado como base nesta dissertação, por ter esclarecido todas as minhas dúvidas de forma rápida e sucinta.

A todos os meus colegas cujos laços criei na universidade e que me acompanharam durante estes cinco anos. Foram sem dúvida uma das peças mais importantes na minha vida e contribuiram para que estes fossem os melhores anos da minha vida e sem os quais não seria o homem que sou hoje.

Um agradecimento especial à minha família, aos meus pais por me terem apoiado em todos os momentos e ao meu irmão, pelos seus conselhos e por me ter acompanhado nesta longa caminhada.

Este trabalho foi financiado pela FCT, *Fundação para a Ciência e Tecnologia*, no âmbito do acordo para a gestão e coordenação do programa UT Austin | Portugal.

Abstract

Polymer processing usually requires several experimentation and calibration attempts to lead to a final result with the desired quality. As this results in large costs, software applications have been developed aiming to replace laboratory experimentation by computer based simulations and hence lower these costs. The focus of this dissertation was on one of these applications, the FlowCode, an application which helps the design of extrusion forming tools, applied to plastics processing or in the processing of other fluids. The original application had two versions of the code, one to run in a single-core CPU and the other for NVIDIA GPU devices.

With the increasing use of heterogeneous platforms, many applications can now benefit and leverage the computational power of these platforms. As this requires some expertise, mostly to schedule tasks/functions and transfer the necessary data to the devices, several frameworks were developed to aid the development - with StarPU being the one with more international relevance, although other ones are emerging such as DICE.

The main objectives of this dissertation were to improve the FlowCode, and to assess the use of one framework to develop an efficient heterogeneous version.

Only the CPU version of the code was improved, by first applying techniques to the sequential version and parallelizing it afterwards using *OpenMP* on both multi-core CPU devices (Intel Xeon 12-core) and on many-core devices (Intel Xeon Phi 61-core). For the heterogeneous version, StarPU was chosen after studying both StarPU and DICE frameworks.

Results show the parallel CPU version to be faster than the GPU one, for all input datasets. The GPU code is far from being efficient, requiring several improvements, so comparing the devices with each other would not be fair. The Xeon Phi version proves to be the faster one when no framework is used.

For the StarPU version, several schedulers were tested to evaluate the faster one, leading to the most efficient to solve our problem. Executing the code on two GPU devices is 1.7 times faster than when executing the GPU version without the framework on one input dataset. Adding the CPU to the GPUs of the testing environment do not improve execution time with most schedulers due to the lack of available parallelism in the application. Globally, the StarPU version is the faster one followed by the Xeon Phi, CPU and GPU versions.

Resumo

O processamento de polímeros requer normalmente várias tentativas de experimentação e calibração de modo a que o resultado final tenha a qualidade pretendida. Como isto resulta em custos elevados, diversas aplicações foram desenvolvidas para substituir a parte de experimentação laboratorial por simulações por computador e consequentemente, reduzir esses custos. Este dissertação foca-se numa dessas aplicações, o FlowCode, uma aplicação de ajuda à conceção de ferramentas de extrusão aplicada no processamento de plásticos ou no processamento de outros tipos de fluidos. Esta aplicação inicial era composta por duas versões, uma executada sequencialmente num processador e outra executada em aceleradores computacionais NVIDIA GPU.

Com o aumento da utilização de plataformas heterogéneas, muitas aplicações podem beneficiar do poder computacional destas plataformas. Como isto requer alguma experiência, principalmente para escalonar tarefas/funções e transferir os dados necessários para os aceleradores, várias frameworks foram desenvolvidas para ajudar ao desenvolvimento - sendo StarPU a framework com mais relevância internacional, embora outras estejam a surgir como a framework DICE.

Os principais objetivos desta dissertação eram melhorar o FlowCode assim como avaliar a utilização de uma framework para desenvolver uma versão heterogénea eficiente.

Apenas a versão CPU foi melhorada, primeiro aplicando técnicas na versão sequencial, e depois procedendo à paralelização usando *OpenMP* em CPUs multi-core (Intel Xeon 12-core) e aceleradores many-core (Intel Xeon Phi 61-core). Para a versão heterogénea, foi escolhido a framework StarPU depois de se ter feito um estudo das frameworks StarPU e DICE.

Os resultados mostram que a versão CPU paralela é mais rápida que a GPU em todos os casos testados. O código GPU está longe de ser eficiente, necessitando diversas melhorias. Portanto, uma comparação entre CPUs, GPUs e Xeon Phi's não seria justa. A versão Xeon Phi revela-se ser a mais rápida quando não é usada nenhuma framework.

Para a versão StarPU, vários escalonadores foram testados para avaliar o mais rápido, levando ao mais eficiente para resolver o nosso problema. Executar o código em dois GPUs é 1.7 vezes mais rápido do que executar para um GPU sem framework em um dos casos testados. Adicionar o CPU aos GPUs do ambiente de teste não melhora o tempo de execução para a maioria dos escalonadores devido à falta de paralelismo disponível. Globalmente, a versão StarPU é a mais rápida seguida das versões Xeon Phi, CPU, e GPU.

Contents

List of Acronyms

ALU Arithmetic Logic Unit

AoS Array of Structures

API Application Programming Interface

AVX Advanced Vector Extensions

CG Conjugate Gradient

CPU Central Processing Unit

CRS Compressed Row Storage

CU Computing Unit

CUDA Compute Unified Device Architecture

DAG Directed Acyclic Graph

DICE Dynamic Irregular Computing Environment

DMA Direct Memory Access

DSM Distributed Shared Memory

DSP Digital Signal Processor

FDM Finite Difference Method

FEM Finite Element Method

FMA Fused Multiply-Add

FPGA Field-Programmable Gate Array

FVM Finite Volume Method

GAMA GPU And Multi-core Aware

GMRES Generalized Minimal Residual

GPGPU General Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HPC High Performance Computing

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

MAGMA Matrix Algebra on GPU and Multicore Architectures

MIC Many Integrated Core

MPI Message Passing Interface

MSI Modified, Shared or Invalid protocol

NUMA Non-Uniform Memory Access

OpenACC Open Accelerator API

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PCIe Peripheral Component Interconnect Express

PDE Partial Differential Equation

QPI Quick Path Interconnect

SFU Special Function Unit

SIMD Single Instruction Multiple Data

SIMPLE Semi-Implicit Method for Pressure Linked Equations

SIMT Single Instruction Multiple Threads

SM Streaming Multiprocessor

SMX Next Generation Streaming Multiprocessor

SoA Structure of Arrays

SPMD Single Program Multiple Data

SSE Streaming SIMD Extensions

List of Figures

Chapter 1

Introduction

1.1 Context

Polymer processing is a pipelined process that usually requires several experimentation and calibration attempts to produce satisfactory results, since different flow restrictions promote unbalanced flow problems. These lab procedures lead to high pre-production costs. Software based simulations have been developed to reduce these costs. One of these simulators is FlowCode [Gonçalves, 2014], which helps to simulate the extruder and cooling phases of the process. FlowCode is also the starting application code for this dissertation work.

In recent years, due to the increasing innovation and performance obtained by the processors, the scientific community also increased the demands, trying to solve more complex and larger dimensional problems which also led to the increasing computing power again. This can be seen as an infinite cycle. However, the process of increasing the computational power is not as easy as it was before when it was "only needed" to increase processor clock frequency. This process reached its end since the processors started to consume too much energy and to produce too much heat. To increase performance, processors with multiple cores appeared with each core running at a slightly lower frequency than their "heavy" single core predecessors. This level of parallelism evolved into multiple multi-core CPUs into one system, each one with its own dedicated main memory (following a Non-Uniform Memory Access (NUMA) architecture) and attaching computing accelerators. These added a few constraints such as their computing architecture, specific programming languages, compilers and memory locality and affinity questions. Programming some of these new devices may require now a different model due to a different architecture. To help developing parallel applications for typical CPUs and also other computing accelerators, libraries and technologies were deployed such as *OpenMP* and *Intel TBB* for CPUs and Compute Unified Device Architecture (CUDA)[1] for Graphics Processing Units (GPUs).

Nowadays, parallelism is the way to go, either by increasing the number of cores of each

[1] http://www.nvidia.com/object/cuda_home_new.html

device or by using multiple distinct devices at the same time - these are called heterogeneous platforms. These platforms contain different architectures, from multi-core CPUs to computing accelerators such as the *NVIDIA* GPUs and the Intel Many Integrated Core (MIC) family of devices. More "exotic" accelerators are also making their way into the main stream, such as Field-Programmable Gate Arrays (FPGAs) and some Digital Signal Processors (DSPs). Deploying applications in these platforms may be a challenge, since each device may have different programming and memory models.

Frameworks have been designed and built to aid the development of applications that efficiently use this mix of computing units. These include StarPU [Augonnet et al., 2009] - the one that gets more international credit - and Dynamic Irregular Computing Environment (DICE) [Barbosa et al.], a more recent one, which provides support for fewer architectures and was previously known as GPU And Multi-core Aware (GAMA) [Barbosa, 2012]. Both have a task-based unified programming and execution model which abstracts the different models/architectures of the underlying devices on heterogeneous platforms. Since all devices have their private memory, these frameworks provide a global memory management and also a workload scheduler, which allows to dynamically assign work to each device at each time step. Although both these frameworks have similar goals, they have differences on their implementation and focus on different techniques to deploy a very efficient application code and execution environment. As heterogeneous systems have different types of devices, it is important to try and use them all to leverage the computing power of these systems. This is also one of the main objectives of these frameworks. Furthermore, the capacity to run on distinct heterogeneous platforms with only one application and to obtain a good performance on all makes these frameworks desirable.

Scientific applications such as the FlowCode, which require heavy computations consuming large amounts of time, are good real case studies to validate the usefulness of these frameworks. The FlowCode application has main two components: a CPU version which only runs in a single core and a GPU version, that due to the parallel nature and high floating point peak performance of the device, runs several times faster. However, the execution of these versions of FlowCode on heterogeneous platforms do not leverage all computing power these platforms provide, since they cannot run on multiple devices at the same time. Moreover, the continuous innovation requirements and the need to solve larger and more complex problems is another difficulty that makes the use of all heterogeneous platform devices even more needed.

1.2 Motivation & Goals

The main contributions of this dissertation work is twofold: (i) to parallelize and improve the efficiency of the original single-core/single-threaded version of FlowCode and (ii) to

analyze and evaluate the frameworks StarPU and DICE, stressing their main strengths and weaknesses and suggesting further improvements to their current status.

It also contributes with an improved version of the FlowCode for execution on multi-core CPUs and MIC accelerators complemented with suggestions of improvements of the GPU version.

The main outcome of this dissertation is an heterogeneous application that simultaneously runs on both CPUs and GPU accelerators. Several conclusions are also discussed on the different scheduler performance with suggestions to further improve all versions. This new version outstands all previous ones by executing faster all tested input datasets.

A note is required to state that the GPU version has a potential to improve its execution time, if some extra time is allocated to better tune the code and associated data structures.

1.3 Dissertation outline

This document has six chapters. Chapter 2 introduces the FlowCode application to be optimized and ported to heterogeneous systems, whereas chapter 3 discusses the technologies used in this dissertation, namely the concept of heterogeneous platforms and the types of devices used in them, complemented by the development frameworks.

Chapter 4 presents the improvements performed on the sequential version and shows the techniques used to create an heterogeneous application using the StarPU framework.

Chapter 5 shows the profiling results obtained with the different versions of the FlowCode with particular focus on various StarPU schedulers and their impact on overall performance.

Conclusions about the work done are presented in chapter 6 which also gives suggestions for future work.

At the start of each chapter, there is an introductory text that shows relevant information and explains the structure of that chapter. This aims to remind the reader of the content of the chapter and allows to search information faster. Similarly, a summary is included at the end of each chapter to quickly resume the relevant outcomes of that chapter.

Chapter 2

FlowCode: numerical modeling of extrusion forming tools

This chapter introduces the main application of this dissertation which is composed of two versions – a single CPU core application and a GPU one. It is a simulator application that helps the design of extrusion forming tools focusing the extruder and calibration phases of the pipeline process and helping to minimize real production costs, by reducing manual and experimental attempts. This chapter explains the methodology employed by the application.

2.1 The Role of the FlowCode Application

Polymer processing is usually done in several experimentation and calibration attempts throughout a pipelined process that consists of several stages, typically extruder, extrusion die, calibration and cooling system, haul-off and saw. The final result is usually not the expected one by the end of the first trial, because different restrictions to flow, among other problems, promote unbalanced flow leading to an irregular profile. Several experimental trials need to be performed until the process and input values are sufficiently tuned. These experiments represent a large cost, mainly because the production process must be stopped to use the extruder to perform tests. However, computer science emerged in almost all areas, including polymer engineering and thus, many experiments passed from laboratorial ones to computer based simulations. The first experimental attempts are performed using applications before being validated in laboratory with the extruder.

The software studied in this work is one of the many applications that have been developed with the aim of solving this kind of problems. This application [Gonçalves, 2014] is able to perform 3D simulations of incompressible flow of newtonian and generalized newtonian fluids (FlowCode will be used for short to represent this application in this dissertation) which are necessary in the extrusion process [Carneiro and Nóbrega, 2012].

Several elements such as fluids (and their properties) are involved in this process. Fluids have various properties which affect the final result of the final processing. For instance, the viscosity property of a fluid affects its resistance to deformation. Newtonian fluids have this value always constant (unless by changing the temperature), while in non-newtonian fluids the viscosity may change. A fashion to model non-newtonian fluids are the generalized newtonian fluids, using the same governing equations and updating the viscosity with some function. This viscosity property can be seen in several examples. Oobleck (cornstarch and water) is a fluid that belongs to the non-newtonian fluids category while water belongs to the newtonian fluids category. While doing different amounts of pressure on water does not change its reaction (it will splash more or less), oobleck will react differently for different quantities of pressure. For less pressure, the oobleck will act as a liquid while for high pressure it will almost react as a solid (its viscosity is different). This is one of several characteristics fluids may have, and that the application needs to have into account in the extrusion process, when computing the flow distribution in the simulation. Otherwise, the flows can be wrongly predicted.

The FlowCode helps calculating the flow distribution at the outlet of the extrusion die by tacking into account the properties of the fluids used and the environment where they are processed. The different velocities in several sections at the outlet indicates an unbalanced flow distribution which leads to an incorrect profile. Therefore, several changes to the die geometry must be performed to achieve more constant velocities and a uniform flow distribution.

[Gonçalves et al., 2013] describe the main characteristics of the application and verify the correctness of the results obtained using three typical benchmarks. They also show the usefulness of the application to improve the flow distribution of a medical catheter.

2.1.1 Governing Equations

Depending on the assumptions made about the type of fluids used and their properties, several equations must be taken into account to describe the fluid flow. The main equations used are the Navier-Stokes equations which describe the motion of fluid substances. Other equations are used to represent the particular characteristics of non-isothermal and non-Newtonian fluids. These equations can be different depending on the fluids physical assumptions that are made but are generally based on the "convervation" of "momentum".

Momentum Conservation Equation

This equation is derived from "Newton's Second Law" and states that "the rate of change of momentum of a fluid particle equals the sum of the forces on that particle".

$$\frac{\partial(\rho u_i)}{\partial t} + \frac{\partial(\rho u_j u_i)}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial \tau_{ij}}{\partial x_j} \tag{2.1}$$

where ρ is the density, u_i the i^{th} velocity component, p the pressure and τ is the stress tensor which is calculated by another function and depends on the type of fluids and their viscosity property η.

As stated before, the viscosity of each fluid may depend on its temperature and is given by viscosity models such as "Power-law" and "Carreau".

Mass Conservation Equation

This equation is also called "continuity equation" and for incompressible flows, is given by:

$$\frac{\partial u_i}{\partial d_i} = 0 \tag{2.2}$$

This equation states that the system mass must remain constant over time if mass is not added nor removed from the system. Therefore, the mass remains the same over time.

Energy Conservation Equation

This equation states that the total energy of an isolated system cannot change, i.e, energy cannot be created nor destroyed.

$$\frac{\partial(\rho c T)}{\partial t} + \frac{\partial(\rho c u_j T)}{\partial x_j} - \frac{\partial}{\partial x_j}\left(k\frac{\partial T}{\partial x_j}\right) = S_u \tag{2.3}$$

where u is the velocity, c the specific heat, k the thermal conductivity (which depends on temperature) and S_u the source terms.

General Differential Conservation Equation

The three governing equations presented may be written as a "General Differential Conservation Equation" for a given property ϕ.

$$\frac{\partial(p\phi)}{\partial t} + \frac{\partial(p u_j \phi)}{\partial x_j} - \frac{\partial}{\partial x_j}\left(\Gamma\frac{\partial \phi}{\partial x_j}\right) = S_u \tag{2.4}$$

where Γ is the diffusion coefficient and S_u is the source term of property ϕ. This general equation is a practical approach to solve the governing equations and comprises four terms which are respectively, the *unsteady* term, the *adjective* term, the *diffusive* term and the *source* term.

Boundary conditions are needed to solve differential equation problems - such as this one - in addition to initial guesses if a iterative solver is used.

2.1.2 Boundary Conditions

These are conditions imposed to simulate the properties and behavior of the system's boundaries. Gonçalves describes the possible boundary conditions existent, which are initially set according to the type of simulation intended simulation.

For the momentum conservation and continuity equations, the boundary conditions are:

- **Inlet** : Imposed velocity - $u_i = u_i^0$
- **Outlet** : Null normal gradient - $\frac{\partial u_i}{\partial x_j} n_j = 0$
- **Wall** : no-slip condition - $u_i = 0$
- **Symmetry** : imposed velocity just with tangential component - $u_i n_i = 0$

For the energy conservation equation, the boundary conditions are:

- **Imposed temperature** : $T = T^0$
- **Insulated and symmetry** : $\frac{\partial T}{\partial x_i} n_i = 0$
- **Natural convection** : $k \frac{\partial T}{\partial x_i} n_i = \alpha(T_\infty - T_f)$ where α is the convection heat transfer coefficient at face f, T_∞ is the environment temperature and T_f the boundary face temperature
- **Contact between two domains** : $k \frac{\partial T}{\partial x_i} n_i = h_f(T_{f1} - T_{f2})$, where h_f is the heat transfer coefficient at face f, T_{f1} and T_{f2} are the temperature of the contact faces $f1$ and $f2$, respectively.

2.1.3 Discretized System of Equations

The equations that govern the flow and energy transfer were presented leading to equation 2.4. The next step is to discretize the equation over each control volume of a mesh - which models the physical domain of the problem to solve - at each time step using one of several methods available. This allows to represent continuous and infinite models into discrete ones, which can be finitely represented. The most common methods are the Finite Element Method (FEM), Finite Difference Method (FDM) and the Finite Volume Method (FVM). The FlowCode uses the FVM method as it is usually used to derive numerical approximations for unstructured meshes.

Meshes are composed by two parts, the geometry and the topology. The former represents the positions of each element (x, y, z in the case of 3D meshes) while the latter represents how each element is connected to each other. For structured meshes, the topology is known and it is given by a regular pattern and thus is not stored in any datastructure. However,

unstructured meshes do not have regular pattern and the connectivity of each node (topology) must be stored in a datastructure. Structured meshes allow an easy improvement and parallelization process whereas unstructured meshes do not, since the indirect accesses (to know the connectivity) of mesh elements result in high cache miss rates.

The FlowCode application uses 3D unstructured meshes since they allow to represent more complex geometries which are used in polymer processing. The ability to refine the mesh is essential since they can provide better results on critical parts of the simulation process. The mesh is stored using several datastructures that comprises three arrays which store faces, boundary faces and cell components of the system. The input meshes are created using GMSH [Geuzaine and Remacle, 2009]. This program allows to generate two and three dimensional meshes, using various different types of elements and faces, such as triangles and tetraedras. The mesh elements are enumerated using gmsh which do not provide a efficient enumeration algorithm, i.e, it may not assure that neighbor elements are enumerated and closely stored in memory which may penalize the application's performance.

Discretizing the "General Differential Convervation Equation" over space and time, using the FVM method, leads to the linear system of equations 2.5, that can be solved by iterative methods.

$$a_P \phi_P = \sum_{nb} a_{nb} \phi_{nb} + S_u \tag{2.5}$$

The FlowCode assembles each term independently, by computing the contributions of each control volume a_P and by performing a summation of the contributions of the cell's neighbors a_{nb}. The *source* term contributions (S_u) come from boundary, pressure gradient, or contributions obtained with values of previous iterations (or initial values). The *adjective* term as well as other contributions like the pressure gradient are computed using the "Least Squares Gradient Reconstruction" method, which uses the gradient computed with information from cell centers and its neighbors. Since the contribution of the cells to the gradient depends only on the geometry, which does not change during the execution, it can be computed only once at each control volume at the beginning of the execution.

However, the equation 2.5 is not enough to solve the fluid flow distribution problem. The velocity and pressure variables must be calculated but, despite the four equations available (three "Momentum Conservation Equation" and one "Mass Conservation Equation") this is not possible due to some constraints of the governing equations. The system presented is non-linear, which means that the unknowns appear as variables of a polynomial of degree higher than one. The "Momentum Conservation Equation" multiplies two velocity components (for inner faces and cell centers) and the pressure-gradient field is also not known.

2.1.4 The SIMPLE

To solve the problem stated, the SIMPLE method was used. This method begins with a guessed velocity and pressure to compute the "Momentum Conservation Equations" obtaining a new velocity at the cell centers. The boundary faces are updated using this new velocity by taking into account the boundary conditions presented in section 2.1.2. At this point, the faces velocities must be corrected which is done by solving a pressure-correction equation. With the result of this equation, the "guessed" pressures and velocities may now be corrected. Algorithm 1 presents the pseudo-code for the SIMPLE algorithm.

Algorithm 1 The SIMPLE algorithm

1: Initial guesses (for velocity ($u^{(0)}$, $v^{(0)}$ and $w^{(0)}$ components) and pressure $p^{(0)}$)
2: Solve momentum conservation equations to find $u*$, $v*$ and $w*$
3: Update inner faces velocities
4: Solve pressure-correction equation
5: Correct pressure and velocities
6: Update CVs viscosity
7: Solve energy conservation equations to find T (temperature)
8: Return to the first step assuming the correct pressure (p) as a guess pressure ($p*$).

The stage 8 of the algorithm happens each time the convergence criteria is not met. The convergence criteria used in the application depends on external residual minimum values for all the systems of equations solved. If all residues are lower than the residual value set by the user, the application stops the execution. Otherwise, a new iteration is performed. Sometimes, the solution is not computed due to slow convergence. For this reason, a maximum number or iteration of the SIMPLE method is imposed by the user.

2.2 Iterative Solvers

Before solving the systems of linear equations, all necessary elements need to be assembled, which includes the matrix **A** and the independent term **b**. Different methods may or may not converge even for the same matrix (depending on the matrix's characteristics, e.g, symmetric, positive definite, banded, etc). For the FlowCode application, there are several possibilities to solve the systems of equations. For the purpose of making a fair comparison between the CPU and GPU application throughout the development of the code, and because the implementation of other algorithm was a difficult challenge to implement on the GPU, the developer chose to solve the equations using the Jacobi algorithm which converge slower than other alternatives.

There are several alternatives that can replace the Jacobi algorithm to improve the application's performance. All iterative methods need to solve a system that have the general form of (2.6) where **A** is a matrix (sparse in the FlowCode application) that stores the

coefficients, $\mathbf{x}$ is the vector solution of the system and $\mathbf{b}$ is the right hand-side.

$$\mathbf{Ax} = \mathbf{b} \tag{2.6}$$

2.2.1 Jacobi, Gauss-Seidel and SOR

The Jacobi method is perhaps the most famous algorithm to solve systems of equations, but it is also one of the slower ones in terms of convergence. This method solves the system of equations where each equation is represented by (2.7).

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^k - \sum_{j=i+1}^{n} a_{ij} x_j^k \right] \tag{2.7}$$

where k is the iteration number and i, the ith equation in the system. The left part of the system contains the contributions of x_i and the righ-hand side contains the other contributions. The algorithm 2 presents the entire method. This method requires an auxiliary array $\bar{x}$ to store the values of the current iterations while the x variable stores the values of the previous iteration. The iterations are independent of each other which allow an easy parallelization of the method. All algorithms presented in this section appear in [Barrett et al., 1994].

Algorithm 2 Jacobi algorithm

```
1: Choose an initial guess x^(0) to the solution x
2: for k = 1, 2, ... do
3:     for i = 1, 2, ..., n do
4:         x̄_i = 0
5:         for j = 1, 2, ..., i − 1, i + 1, ..., n do
6:             x̄_i = x̄_i + a_{i,j} x_j^(k−1)
7:         end for
8:         x̄_i = (b_i − x̄_i)/a_{i,i}
9:     end for
10:    x^(k) = x̄
11:    check convergence; continue if necessary
12: end for
```

Similar and better alternatives are the Gauss-Seidel method and SOR (Successive Over Relaxation). Unlike the Jacobi method, these alternatives converge faster because they use the latest updates from the same iteration as represented in (2.8).

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right] \tag{2.8}$$

Algorithm 3 shows the complete Gauss-Seidel algorithm according to [Barrett et al.,

1994]. Unlike the Jacobi method, the iterations are not independent since each equation depends on all previous computed components (which belong to the same iteration).

Algorithm 3 Gauss-Seidel algorithm

```
1: Choose an initial guess x^(0) to the solution x
2: for k = 1, 2, ... do
3:     for i = 1, 2, ..., n do
4:         σ = 0
5:         for j = 1, 2, ..., i − 1 do
6:             σ = σ + a_{i,j} x_j^(k)
7:         end for
8:         for j = i + 1, ..., n do
9:             σ = σ + a_{i,j} x_j^(k−1)
10:        end for
11:        x_i^(k) = (b_i − σ)/a_{i,i}
12:    end for
13:    check convergence; continue if necessary
14: end for
```

Successive Over Relaxation is similar to the Gauss-Seidel method and differs only by applying a relaxation term in each iteration. Equation (2.9) shows this difference.

$$x_i^{k+1} = (1-\omega)x_i^k + \omega\frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k\right] \tag{2.9}$$

The algorithm 4 show the Successive Over Relaxation method.

Algorithm 4 SOR algorithm

```
1: Choose an initial guess x^(0) to the solution x
2: for k = 1, 2, ... do
3:     for i = 1, 2, ..., n do
4:         σ = 0
5:         for j = 1, 2, ..., i − 1 do
6:             σ = σ + a_{i,j} x_j^(k)
7:         end for
8:         for j = i + 1, ..., n do
9:             σ = σ + a_{i,j} x_j^(k−1)
10:        end for
11:        σ = (b_i − σ)/a_{i,i}
12:        x_i^(k) = x_i^(k−1) + ω(σ − x_i^(k−1))
13:    end for
14:    check convergence; continue if necessary
15: end for
```

2.2.2 Stopping Criteria

All algorithms presented must have a stopping criteria which indicates when to stop iterating. There are several ones, but the most common ones are presented in (2.10) and (2.11).

$$||b - Ax^k||_2 \leq \epsilon \tag{2.10}$$

$$\frac{||b - Ax^k||_2}{||b - Ax^0||_2} \leq \epsilon \tag{2.11}$$

where ϵ is the residual term. The algorithms will stop if this value is less than a predefined one. As can be seen, most operation performed by these methods are matrix-vector products (also known as BLAS 2 operations). These operations are memory-bound and are generally harder to achieve great performance results.

2.2.3 Preconditioned Conjugate Gradient

Another algorithm which is faster than the other ones presented is the Conjugate Gradient (CG) method which makes part of the Krylov subspaces algorithm type. This method generates vector sequences of iterates and residuals by using search directions to allow the approximation to the solution. This algorithm can only be used in situations where the matrix is symmetric and positive definite and may be performed by solving the steps presented in algorithm 5 using a preconditioner M.

Algorithm 5 The Preconditioned Conjugate Gradient Method

1: Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
2: **for** $i = 1, 2, ...$ **do**
3: **solve** $Mz^{(i-1)} = r^{(i-1)}$
4: $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
5: **if** $i = 1$ **then**
6: $p^{(1)} = z^{(0)}$
7: **else**
8: $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
9: $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
10: **end if**
11: $q^{(i)} = Ap^{(i)}$
12: $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
13: $x^{(i)} = x^{(i-1)} - \alpha_i p^{(i)}$
14: $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
15: check convergence; continue if necessary
16: **end for**

Each iterate is multiplied by a multiple (α) of the search direction vector $p^{(i)}$ while the residuals of each iteration are computed as: $r^{(i)} = r^{(i-1)} - \alpha_i p^{(i)}$. The CG method stops if

this value is smaller than the predefined one.

2.2.4 GMRES

The Generalized Minimal Residual (GMRES) algorithm was developed by Yousef Saad and Martin Schultz [Saad and Schultz, 1986; Saad, 2003] and contrary to the CG method, this has purpose of solving nonsymmetric system of linear equations. The most important drawback of this method is that the storage required per iteration increases linearly. For this reason, this method must be "restarted" at each "n" iterations. The last results obtained are the initial ones for each restart. The method may perform excessive work and use more storage for a large "n" and may take some time to converge (or even fail) if "n" is to small. Saad and Schultz state that choosing the best "n" is a matter of experience (and that no rule exists to compute this value). Algorithm 6 shows the preconditioned GMRES method. For more information about iterative solvers, refer to [Saad, 2003; Barrett et al., 1994]

Summary

This chapter introduced the case study, an application which performs the simulation of incompressible flow of newtonian and generalized newtonian fluids for extrusion purposes. This application aids the forming tools creation process which is used on polymer production. This is done by solving the SIMPLE iterative method which relies on several linear systems of equations to compute the velocity, pressure and temperature components of the governing equations. The main purpose of this application is to minimize costs of production, by lowering the number of manual experiments.

Algorithm 6 The Preconditioned GMRES(m) Method

$x^{(0)}$ is an initial guess
for $j = 1, 2, ...$ **do**
    solve r from $Mr = b - Ax^{(0)}$
    $v^{(1)} = r/||r||_2$
    $s := ||r||_2 e_1$
    for $i = 1, 2, ..., m$ **do**
        solve w from $Mw = Av^{(i)}$
        for $k = 1, ..., i$ **do**
            $h_{k,i} = (w, v^{(k)})$
            $w = w - h_{k,i} v^{(k)}$
        end for
        $h_{i+1,i} = ||w||_2$
        $v^{(i+1)} = w/h_{i+1,i}$
        apply $J_1, ..., J_{i-1}$ on $(h_{1,i}, ..., h_{i+1,i})$
        construct J_1, acting on ith and $(i + 1)$st component
        of $h_{.,i}$, such that $(i + 1)$st component of $J_i h_{.,1}$ is 0
        $s := J_i s$
        if $s(i + 1)$ is small enough then (UPDATE$(\tilde{x}, i)$ and quit)
    end for
    UPDATE$(\tilde{x}, m)$
end for

In this scheme UPDATE$(\tilde{x}, i)$
replaces the following computations:

Compute y as the solution of $Hy = \tilde{s}$, in which
the upper $i * i$ triangular part of H has $h_{i,j}$ as
its elements (in least squares sense if H is singular),
$\tilde{s}$ represents the first i components of s
$\tilde{x} = x^{(0)} + y_1 v^{(1)} + y_2 + ... + y_i v^{(i)}$
$s^{(i+1)} = ||b - A\tilde{x}||_2$
if $\tilde{x}$ is an accurate enough approximation then quit
else $x^{(0)} = \tilde{x}$

Chapter 3

The Computational Environment

This chapter presents the concept of heterogeneous parallel platforms and why they became popular in recent years by showing their main characteristics and the most relevant devices available nowadays. It also shows and explain the StarPU and DICE frameworks, that aid the development of code for efficient use of heterogeneous resources. Each framework has the same goals and uses common techniques, but they also have slightly different approaches for code execution, which are also explained in this chapter.

3.1 Heterogeneous Parallel Platforms

CPUs have always been part of the High Performance Computing (HPC) community. These are general purpose processors that aim to efficiently execute code, independently of the type of problem, data structure and algorithm. Although their performance is fairly good, in recent years, the community turned its attention into massively parallel devices such as GPUs (mostly from *NVIDIA*), originally aimed to execute graphical tasks such as creating and rendering images. These devices have a different architecture and can execute a great number of computations (or floating point operations) per second. Recently, Intel also developed a new massively parallel device to execute a very large number of floating point operations per second, the Intel Xeon Phi in the MIC architecture family. Both these devices can be considered as computing accelerators since they aim to speed up the most heavier parts of an application, usually kernels and well known numerical algorithms.

Nowadays, most desktops and laptops contain CPUs and may contain more than one GPU; even smartphones contain GPUs. An increasing number of supercomputers in the TOP500[1] list have compute nodes that contain both CPUs and computing accelerators, either *NVIDIA* GPU or Intel Xeon Phi. These are considered heterogeneous parallel platforms in this dissertation (see figure 3.1) since they contain several devices from different architectures that can execute code in parallel.

[1] Visit http://www.top500.org/ to see the list.

One of the problems with these systems is that heavy numerical computing tasks are often offloaded to the accelerators, namely, expensive functions that can be (relatively) easily parallelized, while the CPU(s) cores are kept idle most of execution time of the program.

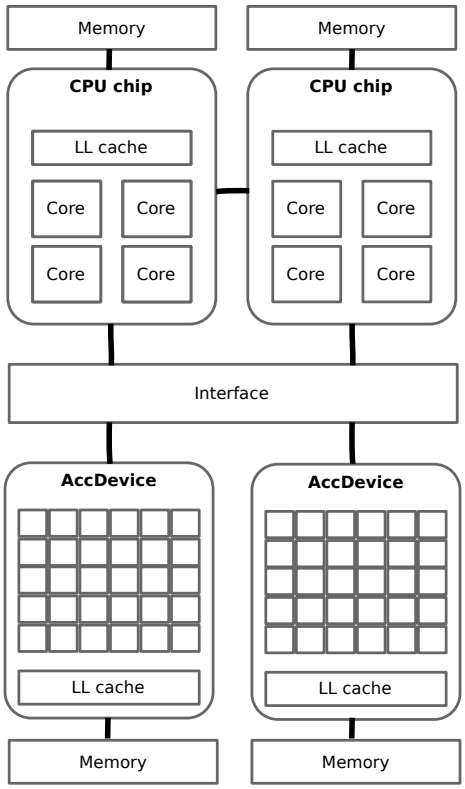


Figure 3.1: An Heterogeneous Parallel Platform

The main challenge is to have programs executing on both CPU cores and accelerators at the same time and further increase the performance of applications. Several frameworks have been created to help solve this challenge such as StarPU and DICE. These frameworks will be discussed further on section 3.4.

To get high performance from these devices is a hard task. There is a great effort trying to reduce the gap between the execution times of the applications and the maximum performance the hardware provides (in heterogeneous platforms, it is the sum of the maximum performance every component can provide).

This is a problem that has not been solved yet due to many reasons. Accelerator devices introduce many hardware constraints, most of them related to memory limitations, execution model and the fact that accelerators and CPUs do not share the same memory (the communication interface showed in figure 3.1 is usually a bottleneck). These limitations do not allow a traditional multi-threaded approach. Furthermore, because accelerators feature their own instruction set and a specific execution model, the code offloaded to these co-processors must be generated by a specific compiler and may be completely different from the code compiled for the regular cores (this is not the case for the Xeon Phi accelerator).

Moreover, moving data from main memory to "device" memory takes a significant time and it is another critical task the frameworks must provide.

Although some less positive aspects have been discussed, the creation and use of specialized hardware such as accelerators or coprocessors is interesting and needed, since it serves to overcome limits faced by the traditional processor microarchitecture.

3.2 Computing Accelerators

The two most popular accelerators were introduced in the previous section, the NVIDIA GPU and the Intel Xeon Phi which had become popular in the HPC community (see the past TOP500[2] list). These will be detailed below.

3.2.1 GPU and CUDA Programming Model

The most known GPUs are from *NVIDIA*, although AMD also produces this type of accelerators. Both have a similar architecture with the difference that *NVIDIA* GPUs can be programmed using either CUDA or *OpenCL*, while AMD GPUs can only be programmed using *OpenCL*. This dissertation focus only on *NVIDIA* GPUs and the CUDA programming model since they are the most used actually. Therefore, all GPU references hereafter are relative to *NVIDIA* GPUs.

These devices have a different architecture and programming model, which means that they require a different programming approach which is known as CUDA C – a C extension. The GPUs were previously used only in graphics processing, mainly in video content and games. But their great computing performance, namely in vector operations, lead to a new generation of GPUs called General Purpose Graphics Processing Units (GPGPUs). Nowadays, there are two version of GPUs, one aiming graphics boards and the other one aiming high performance computing – known as *Tesla* GPUs.

Before presenting the CUDA programming model, the GPU architecture is presented and detailed since it hugely differs from "normal" architectures. At the time this document was written, there are three generation of high performance computing GPUs, namely *Tesla*, *Fermi* and *Kepler*. They all have similar characteristics but they differ mostly in terms of number of simple cores and capacity. The latter is the more powerful device available. As one of the devices used in this dissertation is a *Kepler*, its architecture is explained along with the *Fermi* to compare them and show their main differences.

Each device contain a variable number of Streaming Multiprocessors (SMs) for *Fermi* or Next Generation Streaming Multiprocessors (SMXs) for *Kepler*. The architecture of the *Kepler* device is shown in figure 3.2.

[2] http://www.top500.org/

[3] http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Figure 3.2: Kepler Architecture – full chip[3]

The figure shows 15 SMXs that share a level 2 cache and that are connected to memory by 6 memory controllers that allow a great memory bandwidth.

The architecture of a SMX can be see in figure 3.3. Each SMX contains 192 single-precision floating-point compute elements (more commonly known as CUDA cores). Apart from that, they contain 64 double-precision units, 32 Special Function Units (SFUs), and 32 load/store units. They can provide a little more than one Teraflop of double precision peak performance and almost four Teraflops in the case of single precision. NVIDIA claims that the main performance difference between the single and double precision operations is that single precision peak performance is twice the double precision one (in *Fermi*) and next to 3 times faster in Kepler.

There are also 4 warp schedulers and 8 dispatch units which provide work to each SMX. More information about this terminology will be explained further on with the CUDA programming model.

The GPUs execute CUDA kernels which are parallel implementations written in CUDA C. A CUDA kernel is executed by a *grid* or array of threads (where each thread will be executed by a CUDA core). All threads in a *grid* will run the same kernel code following a Single Program Multiple Data (SPMD) approach. Each thread has indexes that are used to compute memory addresses and make control decisions. The thread array (*grid*) is divided in thread blocks that can contain a maximum of 1024 threads each in the last CUDA version. To launch a kernel, both *grid* and thread blocks numbers must be provided.

The execution of a kernel will generate several thread blocks that will be assigned to the

Figure 3.3: The SMX Architecture in Kepler

SMXs. Each block is further divided in sets of 32 threads called *warps* that are scheduled and executed via the *warp* schedulers and dispatch units of each SMX which are showed in figure 3.3. Each SMX can execute 4 *warp* (each *warp* executing two independent instructions per clock-cycle).

Each SMX core has a private L1 cache (with 64 KB) and a read-only cache (with 48 KB). There is also the presence of a "shared memory" but this memory shares the physical L1 cache memory. The user can configure the L1 cache memory to be:

1. 16 KB share memory and 48 KB L1 cache
2. 32 KB share memory and 32 KB L1 cache
3. 48 KB share memory and 16 KB L1 cache

Previous versions of GPUs could only use configurations number 1 and 3. These caches are not designed to reduce memory latency, but to allow high throughput. It is important to notice that threads inside a block can share information using the "shared memory" and threads in other blocks do not see or access this information. Unlike CPUs, GPUs typically do not have branch prediction and data forwarding and the CUDA cores ALU's are heavily pipelined. The "Kepler" K20 has a maximum peak double precision floating point performance of 1.17 Tflops and 3.52 Tflops for single precision according to the NVIDIA "Kepler" family datasheet[4].

In short, CPUs are best used in situations where latency matters, while the GPUs should be used where parallelism is important (throughput). While this is true, having processing power idle is not convenient and this is why the frameworks addresses this important topic.

For more information about *Kepler* and CUDA C, refer to [NVIDIA, 2012a,b] and [NVIDIA, 2013a] respectively.

3.2.2 Intel Xeon Phi - MIC

Recently, another type of accelerator was developed by Intel. The Intel Xeon Phi belongs to a new generation of accelerators, following the MIC architecture. These processors contain a large number of cores running at low frequency (1.1Ghz or less). They have the great advantage that it is not needed to learn a new language to program the accelerators, since they share similar features of common CPUs and can programmed using for example, *OpenMP* [OpenMP Architecture Review Board, 2013] , *MPI*, *Intel TBB* or *Pthreads*. The

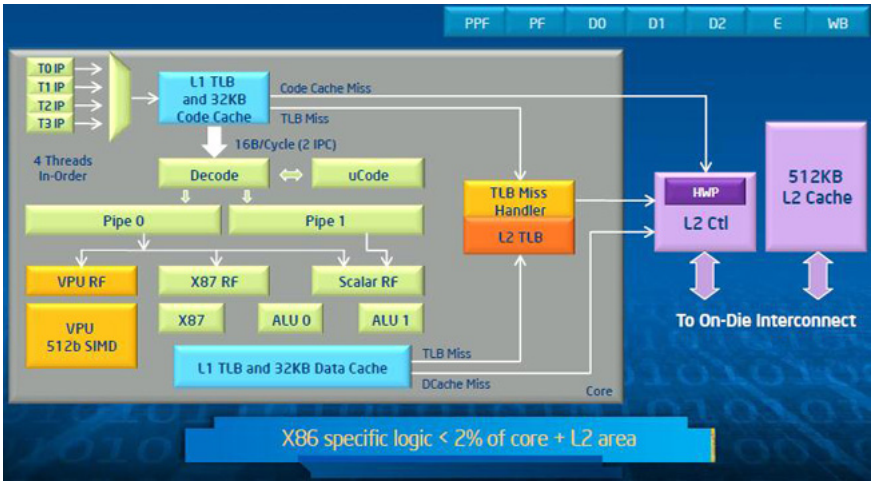


Figure 3.4: Xeon Phi core

more common number of cores is 60/61. As can be seen in figure 3.4, each core has a dual issue pipeline and has support for 4 in-order instructions that shares 512KB of L2 cache. Thus, the total amount of second level cache memory is more than 30MB for the entire

[4] http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf

coprocessor. The more powerful feature of the MIC is the vector processing unit of 512 bits, allowing the execution of 8 double precision or 16 single precision operations in parallel (with a Single Instruction Multiple Data (SIMD) approach). This is a difference over the GPUs. While GPUs follow a SPMD/SIMT approach – that is, the same program is executed in different data elements and each thread executes the same instruction respectively –, the MIC uses a vector unit that executes the same instruction over different data elements at the same time.

The floating point peak performance is given by:

- 16 (SP SIMD) x 2 (FMA) x 1.238 (GHz) x 61 (# cores) = 2.416 TFLOPS for single precision arithmetic
- 8 (DP SIMD) x 2 (FMA) x 1.238 (GHz) x 61 (# cores) = 1.208 TFLOPS for double precision arithmetic

Each core has also support for Fused Multiply-Add (FMA) and together with the vector units, the Xeon Phi has a theoretical peak performance of more than 2 Teraflops for single-precision and more than 1 Teraflop for double precision. These values indicate that the Xeon Phi and Kepler K20 accelerators have a similar peak performance for double precision arithmetic.

Also, it should be noted that the fact that the pipeline issue instructions in-order increases memory related problems. For instance, when there is a stall on a thread, this thread needs to wait for data to be fetched from memory, which generally takes many clock cycles for each stall. Intel overcome this problem with hyper-thread 4-ways. Thus, when one thread is waiting for data, another one may issue instructions.

3.3 Efficient Data Representation and Processing

The previous section showed that accelerators have different objectives and characteristics from the ones of a CPU. Hence, data representation and processing may need to be different to achieve good performance of each device. Several topics are relevant to address, and these include memory locality, storage format and vectorization.

3.3.1 Memory Locality

Memory locality is relevant for both CPUs and accelerators to achieve good performance. If consecutive accesses to sets of data are in contiguous memory locations, few fetches will be needed to load the data into memory caches. However, memory locality is even more critical to GPUs when threads of a warp try to fetch data. If the 32 threads accesses contiguous data positions, a single fetch operation is needed by coalescing all accesses into a single request. But if threads do not access contiguous positions, more fetch operations may be needed

up to a maximum of 32. This is one of the main reasons why it is hard to achieve good performance on the FlowCode or similar ones (and mostly on GPUs). The memory locality also lead to the importance of data representation on data-structures, being the common ones Array of Structures (AoS) and Structure of Arrays (SoA).

3.3.2 AoS VS SoA and Vectorization

AoS is a collection of contiguous structures. This has the advantage of having the information compactly represented. If some tasks need all data that is in a structure, this representation may be a good fit as all data cache lines fetched will contain only data needed by the task. If the structure would be represented in the SoA format, each line fetched would only contain on element really used by the task (and thus fetching much more lines would be necessary). In this case, using AoS could be the best option. However, with the increasing capacity of vector operations (either in CPUs or accelerators), the SoA may be a better fit. This model gathers several arrays into a structure and thus, operations that are performed in contiguous data of an array can be performed using vector operation (vectorization) leading to a performance increase. The same cannot be done with the AoS model. As Pascal Costanza[56] states, the advantages and disadvantages of each configuration can be summarized as:

- AoS
  - All member of each instance next to each other.
  - Good for data locality, less good for Vectorization.
  - Not good when only a subset of the members is needed.
- SoA
  - Each member of all instances next to each other.
  - Good for Vectorization, less good for data locality. (SSE2, AVX, ...)
  - Good when only a subset of the members is needed.

Therefore, it is hard to devise which one must be used to obtain the best possible performance. A typical decision is to use AoS in CPUs and SoA in GPUs, mostly because of its SIMD architecture and coalesced memory access.

3.3.3 Matrix Storage Format

The matrix storage format is also relevant to allow good performance. If the matrix has many zeros, it does not make sense to store and represent all these zeros which leads to the

[5] https://indico.cern.ch/event/214784/session/7/contribution/475/material/slides/0.pdf
[6] https://github.com/ExaScience/arrow-street

use of matrix representations. The sparse matrix used in the FlowCode is stored using the Compressed Row Storage (CRS) format. There are several different types of storage formats but most of them are basically variants of the same, i.e., storing the non-zero elements of the matrix into a linear array and the row and column indexes into two auxiliary arrays.

The most common ones are:

- Compressed Row Storage (CRS)
- Compressed Column Storage (CCS)
- Block Compressed Row Storage (BCRS)
- Compressed Diagonal Storage (CDS)

All these storage format store the non-zero elements of the matrix in a "values" array. The CRS stores the index of the first element of each row into a "row_ind" array and in a "col_ind" array, the column index in the original matrix of each element of "values". CCS does the opposite, by storing the first element of each column into an array and the row index of each matrix element into another one. The BCRS format is a block version of the CRS format. It is an interesting format for matrices that have regular pattern of dense blocks of non-zeros elements. It consists of three arrays. The first is a two dimensional array that stores the values of the non-zero blocks (of length "n") in a row wise fashion. The zero elements of the non-zero blocks blocks are also stored but blocks that only contain zero elements are not stored. The "col_ind" array stores the column indices in the matrix of the non-zero blocks while the "row_ind" stores the indices of the first block of each block row. This format provides less indirect accesses than the previous ones, which may lead to more efficient computations on the matrix. The CDS format tries to leverage the property of banded matrices which have a more or less constant bandwidth from row to row. The original matrix is stored into another one which only contains the sub-diagonals of each row. This format may also store some zero elements or even sub-diagonals of zero elements.

No storage format can be considered the most efficient one, since the usage of one or another may depend on the type of problem, the platform where the application will run and the characteristics of the matrix.

3.4 Frameworks for Heterogeneous Platforms

Section 3.2 showed the typical accelerators used in heterogeneous platforms as well as their powerful features which makes them essential in situations where high peak floating-point performance is needed. It also showed their main characteristics which makes executing code simultaneously along with CPUs such a hard task. The use of StarPU or DICE may ease this problem.

3.4.1 StarPU

StarPU [Augonnet et al., 2009, 2011] is a scheduling system of graphs of tasks which are assigned to heterogeneous platforms devices. The only thing that is required is an implementation of the code for the different architectures the platform contain and all the rest will be done without the programmer's awareness. StarPU is responsible for two important aspects which are the data management and the workload scheduling policy over the several computing units of a heterogeneous platform.

To manage the data of an application, all data is registered to StarPU. This data will be kept in main memory and will be automatically transferred to the processing units (and also device memory in the case of accelerators), as they are needed by tasks. This is done using a virtual shared memory model with relaxed consistency (to avoid over-synchronization).

To control workload scheduling, StarPU provides several task scheduling policies, which are presented in table 3.1. Most of them relies on a common scheduling strategy and adds a

Name	Policy description
eager	scheduler uses a central task queue, from which workers draw tasks to work on. This however does not permit to prefetch data since the scheduling decision is taken late. If a task has a non-0 priority, it is put at the front of the queue.
prio	scheduler also uses a central task queue, but sorts tasks by priority (between -5 and 5)
random	scheduler distributes tasks randomly according to assumed worker overall performance
ws	(work stealing) scheduler schedules tasks on the local worker by default. When a worker becomes idle, it steals a task from the most loaded worker
dm	(deque model) scheduler uses task execution performance models into account to perform an HEFT-similar scheduling strategy: it schedules tasks where their termination time will be minimal.
dmda	(deque model data aware) scheduler is similar to dm, it also takes into account data transfer time.
dmdar	(deque model data aware ready) scheduler is similar to dmda, it also sorts tasks on per-worker queues by number of already-available data buffers.
dmdas	(deque model data aware sorted) scheduler is similar to dmda, it also supports arbitrary priority values
peager	is similar to the eager scheduler, it also supports parallel tasks (using *OpenMP*)
pheft	similar to the dmda scheduler, it also supports parallel tasks (using *OpenMP*)

Table 3.1: Scheduling strategies implemented in StarPU [INRIA, 2014].

feature which help to improve the scheduling a little bit more. Although the table is almost self describing, more information about these schedulers need to be provided. The "eager" scheduler uses a central task queue (which is stored in main memory) and from which workers

pop tasks to work on. The "prio" is similar to the previous scheduler with the difference of allowing priority tasks. The queue is then a priority queue. The "random" scheduler may seem quite inefficient and it can be on most situations but it has the advantage of not making computation to know to which device it needs to send data. It only generates a random number (based on power/speed of the device) with the purpose of assigning a higher number of tasks to the more powerful devices. Sometimes however, even when distributing tasks to the workers according to their speed may not be the best option, mostly because tasks may not be equally expensive. The schedulers that start with**dm** are based in the "dm" scheduler – which performs some calculations to know to which device to send information for the task termination to be minimal. This model can be improved by taking into account in which device the data required by a task is (and a possible data transfer time) – and decide whether it is better to move data or to migrate the task to another processing unit [Augonnet et al., 2010a] – , or even having queues in each device. A characteristic of all these schedulers is that they can only push tasks to a queue when there are no dependencies left. The "peager" and "pheft" are experimental and may not be as stable as the other ones. They are based in the "eager" and "dmda" respectively but allow a task to be executed in parallel in CPUs using *OpenMP* while the other schedulers execute a task in each core (or devices in the case if GPUs). Some schedulers such as "dm" support performance models. These models allow to estimate the duration of a task before it was executed, and thus allow StarPU to achieve a good scheduling policy. To have these models, StarPU must execute several times (10 executions are recommended) to gather accurate data. StarPU extensibility allows to the user to define a new scheduling policy by defining several methods, a method called at the initialization of StarPU, and the push and the pop methods that implement the interaction with the abstract queue. For more information about the scheduling policies, refer to [INRIA, 2014].

Globally, the framework can be seen as an application that perform pushing tasks to queues (which are controlled by a scheduler) and then are sent to processing units. Sometimes, data structures of one kind can be efficient on CPUs but not on GPUs. StarPU has a great feature which allows to have different data structures for CPUs and GPUs and it can convert them (only some schedulers are optimized for this feature). This conversion is basically a task but it is managed by StarPU internally.

In a programming perspective, tasks consists of a codelet (a function that contains function implementations for several architectures) working on a list of handles, which are interfaces to data that is managed by StarPU [Hugo et al., 2013; Agullo et al., 2010]. The access mode (e.g., read-write) of each handle is also required so that the runtime can compute the dependencies between tasks. Unregistering the handles causes that everything goes back to the global memory.

StarPU tasks are asynchronous and submitting a task is a non-blocking operation. When StarPU knows that for one device, data will not change with the next tasks, it will send

information to all other devices in the system. This works similarly to write-trough caches. If information keeps changing constantly, and information is sent to other devices memories, this would generate useless memory traffic.

StarPU also has the advantage of giving the user two possible ways of using the framework, either by using the C extension or by using the StarPU API [Courtès, 2013]. The former provides a quick and easy way to execute and application on a heterogeneous platforms, while the latter is more powerful and can be used by programmers with more expertise to further improve the performance of their applications.

Resuming, there are several steps to be able to port an application to StarPU. These are:

- Register the different data to StarPU
- Create wrappers for the different kernels implementations
- Describe the algorithm as a set of tasks and submit them.
- Task dependencies are either given explicitly or automatically derived from data dependencies if the different tasks are submitted in an order that corresponds to a valid sequential execution.

Moreover, to avoid unnecessary transfers, StarPU keeps data where it was last needed, even if was modified there, and it allows multiple copies of the same data to reside at the same time on several processing units as long as it is not modified. StarPU tries to improve performance by overlapping data transfers with computation [Augonnet et al., 2010b].

StarPU can create a graph of task dependencies using several mechanisms. The first one which does not require a large effort for the developer, is created by using "permissions" for each handle (which represent at a higher level the program data such as arrays or variables). When submitting a task, each data handle of the task will be assigned a permission, **R** (read-only), **W** (write-only) and **RW** (read-write) and StarPU will automatically compute the dependencies between tasks. Another way to let StarPU know of task dependencies is by using the callback mechanism. For example, tasks that must be executed after "TaskA" will be launched by the callback function of "TaskA". Callback functions are always executed in the CPU which is why they do not perform heavy tasks but launch other tasks instead. A third way to perform the same operation is by using tags. When creating a task, each one can be assigned a tag and also an array of tags representing tasks that need to finish executing before the task can start execution.

StarPU also allows to define an interface to represent the data-structures. There are already some predefined ones such as for vector structures. The same happen for filters. For each interface, StarPU has several filters to split data into smaller pieces that can be executed by tasks concurrently.

3.4.2 DICE

The previous framework is maybe the reference in the field. It was built throughout several year by researchers at the University of Bordeaux and Inria[7]. DICE is a relatively recent alternative that was born from a partnership between elements from the University of Minho and University of Texas at Austin. This framework claims to have a strong point which is great efficiency in irregular algorithms and applications – at least based on several results presented in [Barbosa et al.].

As opposed to StarPU which has many different task scheduling possibilities (plus the ones the user can create), DICE only has two:

- **Static** - the framework will use a configuration file which explicitly contains the load of each device. This can be useful to calibrate by hand or by using a script.
- **Adaptive** - the framework will assign chunks of the workload (jobs/tasks) to the devices depending on the current state and the job execution history and work stealing will also be enabled.

The static scheduler reads a file that contains numbers representing the percentage of work each device may process, and uses them to assign tasks. Devices that have a larger number will have more tasks to execute.

The adaptive scheduler and the ability to perform *dicing* is what makes DICE a strong alternative. *Dicing* has several purposes and is used in several situations. This operation allows to split tasks into fine-grained ones, which can then be assigned to devices. The various devices of a heterogeneous platform may have certain restrictions in terms of the size of tasks that it can process. GPUs will tend to process coarser tasks whereas CPUs will tend to process finer ones. Therefore, the *dicing* function may be used to split tasks accordingly to the device capacity by beginning with a coarse task and recursively divide it, improving work-sharing distribution.

There are two levels of *dicing*. The first level has the objective of splitting the job as best as possible to all available devices. In the second level, the *dicing* function has the purpose of splitting the work to available cores (i.e., in a CPU). The *dicing* function can be applied the split tasks but currently, the opposite (that is, gather work to create coarser tasks) is not possible, because an efficient way to do so have not be found and implemented. Most of the time, *dicing* is used to split work across available devices.

A great feature of DICE and its adaptive scheduler, is the ability of work stealing. When a device has no more tasks to execute, it will try to stole work from other devices. This technique is essential to provide a good performance on recursive and irregular algorithms. This feature can also be found in the Intel TBB[8] parallel library. The work stealing only happens at the device level, i.e., there is no work stealing inside a device such as a CPU.

[7] Visit http://www.inria.fr/

[8] Visit https://www.threadingbuildingblocks.org/

This mechanism works as follows: a thread from the device without work will communicate with the "superthread" - the thread that knows how much work each device has - and will ask for work. The superthread will watch in its table and answer with the device that has more work to do. The thread will steal work from that device.

It may be needed to use the *dicing* function after work stealing. This happens most of the time when a CPU/core stoles work from a GPU.

To further improve efficiency, synchronization barriers are used with timers which know how much work has been done and how much free time each device had. In the next task distribution, work sharing will be readjusted.

In short, efficiency results from two mechanisms: the synchronizations barriers which allows to improve the workload distribution and the work stealing which reduce task execution time while workload distribution balance has not been achieved.

DICE and StarPU have a similar memory model which is characterized by a weak consistency model. Without it, programs would perform poorly, since every operation in shared data would need a synchronization operation. To accomplish this, DICE keeps track of "storage classes" data copies within accelerator embed memories and features a data prefetching engine.

There are three types of "storage classes" in DICE [Barbosa et al.]:

- **Read-only (RO)** - As the data is read-only, every device can have a copy of it and no synchronization operations are needed. There is no write operations in global memory, which means the the data is always consistent and thus, every device can have a copy of it.

- **Device-exclusive (DE)** - Only one device at a time has access to this page. Tasks that needs this page (but dot not have access to it yet) will be delayed.

- **Pinned Memory** - Every write operation requires synchronization and happens in the global memory. These pages are shared across all available devices. Application performance decreases tremendously when using these pages.

These classes are needed to provide a global memory management model so that each device may see the global memory and DICE may provide synchronization. One of these classes must be associated to each defined task. RE are efficient, since there are no synchronization operations associated with them and DE pages are similar to locks – only one task may work on the DE data at each time. While these pages can be used without large costs, the use of pinned memory pages should be avoided as operations executed in these pages are synchronized. Part of the global memory will be kept in each device for efficiency reasons and all RO and DE pages from a task will be copied to the device for task execution. Recall that a task can only be executed when it is possible to copy memory pages from global

memory to device memory. This is only true when a task is executed in a device whose memory is not the global memory.

3.4.3 Similarities and Differences

Both frameworks have the same purpose of executing applications in all resources of an heterogeneous platform. There are similarities and differences between the two to attain this same goal, which have been presented in past sections. This section quickly summarizes several of them.

Both frameworks may change task execution order. This increases somewhat the complexity of using these frameworks. Once data is registered, the application does not access it anymore through its memory but through an abstraction (handles in case of StarPU and SmartPointers in the case of DICE). Both frameworks can perform asynchronous transfers and execute asynchronous tasks and they rely on performance measurements from previous executions to better distribute tasks, optimizing policies incrementally. Each task/work job must have an implementation for the device targets (CPU and GPU), although this is not strictly the case regarding DICE, which allows to use the same code implementation for running in both platforms/devices. Both frameworks require a CPU core to control a GPU device.

They both have support for external frameworks/libraries such as MAGMA and Intel MKL [Intel, 2013c,a,b] and use a relaxed consistency model that avoid constant synchronization operations. For extreme performance, they need to guarantee a great balance between task distribution and data transfer.

StarPU gives the programmer the choice of using the "C extension" or the API. This can be useful for programmer that do not have great experience. On the other hand, and although DICE may actually be more challenging to use, the fact the it does not (optionally) require code written in CUDA can be helpful.

StarPU offers a large number of task scheduling policies by incrementally adding features to a set of core schedulers, which result in an improved work balancing. DICE only offers two schedulers and the adaptive one provides both good work balancing and work stealing, a characteristic that is also used in other frameworks for irregular algorithms.

Summary

This chapter explained the key concepts of heterogeneous platforms, and also the GPU and MIC devices that are the core of these platforms. To help lowering the hard work that is needed to create applications that fully run on heterogeneous platforms – applications

that do not offload heavier parts to the accelerators, but instead run them on all available resources at the same time – several frameworks were created as DICE and StarPU. These share the same objective but have some differences in their approach.

Chapter 4

MultiFlowCode: Efficiency Improvements

This chapter describes the several techniques used to improve and parallelize the CPU version, and also addresses some issues of the MIC and GPU versions. The steps taken to create the StarPU implementation are also explained, which take into account the limitations and features of both the FlowCode and the framework.

4.1 FlowCode CPU

There are several ways to improve an application such as performing sequential or parallel improvements. To perform sequential improvements, the profiling of the application has been done. This is useful to know the main bottlenecks of the application and to evaluate which part of the application deserves a deeper attention. The profiling was performed using GNU gprof, with the help of the gprof2dot[1] application to allow the graphic visualization and interpretation of the profiling information. The profiling was done using one of the input datasets available, with a number of elements ranging from three hundred thousands to two million elements. This and other datasets are used to evaluate the benefits of the optimization techniques in chapter 5. The analysis showed that the application spends most of its time in two functions that are called from the SIMPLE function, the main function of the application. The time spent on the system of equations solver (in this case, Jacobi) and the get_gradient function as can be seen in figure 4.1. The jacobi function takes between 20-25 percent of total execution time, whereas get_gradient takes more than 40 percent. However, there is a great difference between the two functions, which is the number of time each one is called. This means that one call of the Jacobi function takes large amounts of time when compared to the get_gradient function. The number of times this function is called indicates that one possible problem for consuming large amounts of time is the

[1] Visit https://code.google.com/p/jrfonseca/wiki/Gprof2Dot for more information about this tool.

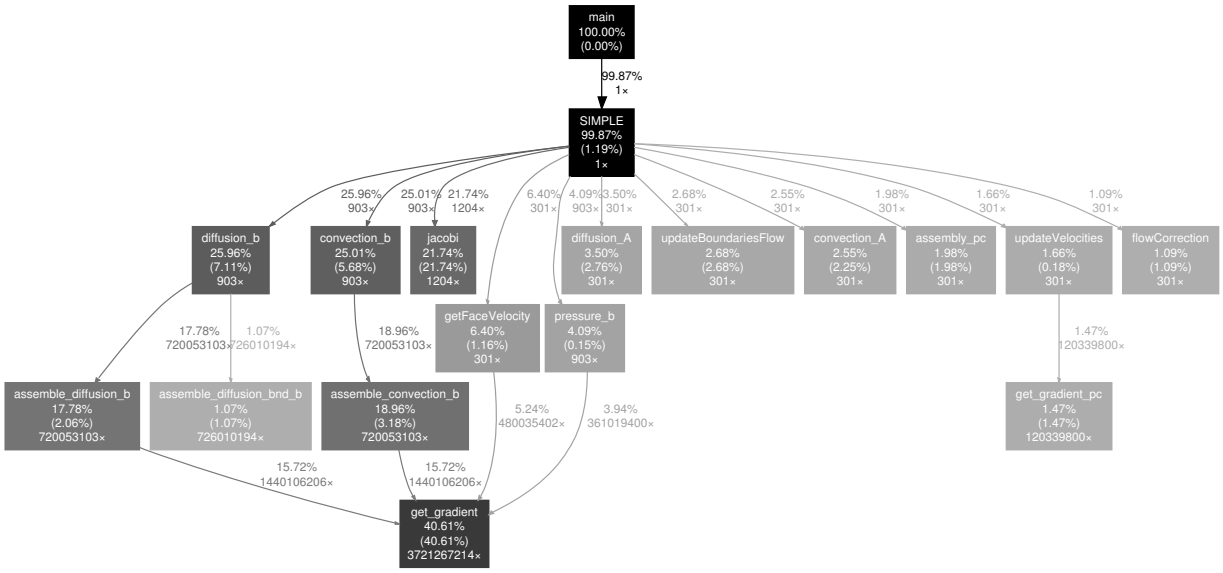


Figure 4.1: FlowCode profiling

function calling overhead. To improve this situation, the get_gradient and other functions have been inlined. This means that the overhead for pushing and poping things out of the stack disappears, greatly reducing the function's execution time. This function was also called a large amount of times to perform computations that would return the same values frequently, when those values only change for each different velocity component. Hence, they were computed only once per system of equation at the beginning of the assembly phase and stored on a datastructure to be used by the other assembly functions. Other values were also calculated many times even though they remained the same throughout all the execution. These values have been computed before the SIMPLE algorithm and stored in a structure to be used in the algorithm.

It is important to notice that many other functions have been omitted from the figure due to the fact of their execution time being residual. The figure also helps to see what is happening in the code, showing which functions calls which. The get_gradient function is called from many functions such as the assemble functions which have the purpose of gathering the necessary elements for the several system of equations. Among these, the diffusion_b and convection_b functions - which compute the diffusive and convective term - seem to take significant time.

The FlowCode's datastructures follow an AoS approach that allows more locality for each element, typically more important for CPU implementations as was explained in section 3.3.2 (although the larger SIMD operations available in CPUs nowadays may be a good fit for a SoA approach). It has three main structures responsible for the solver properties, the systems of equations to solve and the mesh. Other structures are used to help representing the mesh, faces and cells, namely:

- **Vertex** This structure stores the x, y and z coordinates of a vertex.

- **Vector** Structure similar to the "Vertex" but adds the length of the vector.
- **Face** Structure that stores the velocity at the center of the cell, its center coordinates, area and other data. It also keeps the location of the two neighbor cells which can be accessed in the cells datastructure.
- **Boundary face** Structure similar to the "Face" structure. It contains the boundary type and an array of the properties presented in chapter 2 such as temperature, velocity (3 components), pressure, etc.
- **Cell** Contains geometrical properties such as the cell center. It also contains several properties - most of them equal to the "Boundary_face" structure - such as the diffusivity, viscosity, temperature, pressure and velocity components. The properties of both structures are changed in each iteration of the SIMPLE method.
- **Mesh** Main structure of the FlowCode application. It contains arrays of vertices, faces, cells and boundary faces (these last three are updated at each iteration of the SIMPLE method).
- **Matrix** This structure represents a matrix in the CRS format.
- **Solver** Simple structure that contains several solver properties such as maximum number of iterations for each system of equations and their its residues.

As stated in chapter 2, SIMPLE is an iterative algorithm which solves, for each iteration, several systems of equations and performs operations such as updates on some of the datastructures presented above. Some of these operations were performed in application the same way as they were devised. Hence, several function and loop iterations were used to perform operations that could be done in only one loop iteration. In every place where two or more loop traversals of the same structure could be done at the same time, these have been replace by one. These are called loop fusion and loop reordering techniques, which can make the code less readable but allows a faster execution by reducing the loop traversal overhead. An overview of the main functions and tasks of the SIMPLE method are presented next along with an explanation of their purposes and the loop techniques applied to improve their performance.

Assemble Matrix A for velocity

This task comprises three functions responsible for assembling the "diffusion", "convection" and "unsteady" elements of the system of equations. The first only iterates over the face and boundary face arrays, and for each face, it stores in the matrix, the contributions of the two neighbor cells.

The fact of contributing to two cells at the same time poses a problem when more than one thread is executing. This could lead to threads updating the same value, but only one of those succeeding in this operation leading to a wrong result. There are several ways to surpass the problem which will be explained in section 4.1.2. The function responsible for assembling the convection element also iterates through the face and boundary face arrays and thus, also suffer from the same problem. Since both functions iterate over the same structures, these have been replaced by one function performing only a loop traversal. The third function iterates through the cell array and for each cell writes in the diagonal position of the matrix.

Assemble Rigth Hand Side for velocity

Four functions are used to assemble the contributions to the right-hand side of the system, namely the "diffusion", "convection", "unsteady" and "pressure" contributions. The first two are similar with the ones used when assembling the matrix **A**. Thus they have been replaced by one function which iterates though the faces and boundary faces only a single time. However, they also suffer from the problem of contributing to the two neighbor cells a the same time if multiple threads are used. The other two functions iterates through the cell array and can be parallelized safely. Since they iterate through the same structure, these have also been replaced by one function only.

Jacobi

This function solves the different system of equations. It is easily parallelized since this method computes mostly matrix-vector operations.

Update Boundaries Flow

This task is comprised of two function that update the velocity and pressure components of the boundary faces and depend on the boundary type. Again, these have been replaced by one function that iterates through the structure once, performing both tasks at the same time.

Flow Correction

Corrects the velocity components of each boundary face. This function has been merged into the previous one since they iterate through the boundary faces.

Get Face Velocity

This function updates the velocity of each face with velocity contributions coming from their two neighbor cells.

Assemble Pressure Correction

This task contains functions which iterates through all faces and boundary faces creating a new matrix (using the diagonal values of the velocity matrix) and a new right hand side with the updated boundary properties. Both functions are similar with the tasks "Assemble Matrix A for velocity" and "Assemble right hand side for velocity", and thereby, contain the same problem of writing in multiple places at the same time (when using multiple threads). Loop fusion have also been applied here. The pressure-correction system is then solved using the "Jacobi" method and the solution is used to correct the velocities of faces and cells.

Correct Face Velocities

This function corrects the velocities of each face using the diagonal of the pressure-correction matrix.

Update Velocities & Update Viscosity

Both tasks update the properties for each cell using the diagonal of the pressure-correction matrix.

4.1.1 Iterative Solvers

As the Jacobi iterative solver was also taking a significant amount of time, this solver has been replaced by other other ones that allow faster convergence [Saad, 2003]. For the velocity equations, the method used was the GMRES method but other ones could be used such as the BiConjugate Gradient method. For the pressure equations, which provides a symmetric system of equations, the CG method was used.

The Intel MKL library has been used to implement both solvers. These solvers are based in reverse communication - the library contains these solvers and may pass the control of the solver to the user which can on its turn perform control operations. This means the user can apply several different preconditioners and may change residual error values and the maximum number of iterations while the algorithm is running. The diagonal preconditioner was used to improve convergence of the system.

4.1.2 Coloring Scheme

As stated, parallelizing the assembly function requires a slightly different approach, since multiple threads may write data from the mesh to the same positions of the matrix when assembling data from the faces and boundary faces. This happens because each face contributes to two positions on the matrix. In the sequential code, the single thread can add the contributions to the two cells in the matrix at the same time, but when two or more threads

performs this same strategy they may write in the same memory position, overwriting the correct result. To solve this problem, a coloring scheme method was used. There are other alternatives to solve this type of problem, such as performing a cellwise approach. With this approach, there are only iterations through the cells arrays and each thread is responsible for assembling the contributions to its own cell. Unlike with the coloring scheme, where the values computed in each face and boundary face contribute to two cells at the same time, this approach needs to perform twice the computations but offers more parallelism and may provide better results [Kampolis et al., 2009]. This approach has not been implemented since it would require changing significant parts of the code for both CPU and GPU at an already advanced stage of the dissertation.

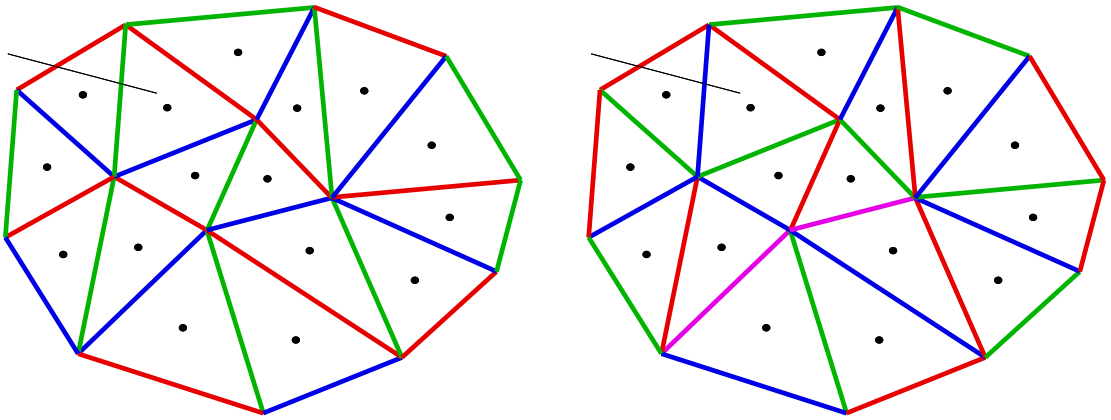
Therefore, before executing the SIMPLE method, a mesh coloring phase occurs, which was the approach taken in the GPU implementation. This is done by iterating through the faces of each cell ensuring that each one has a different color. This same process is applied to the boundary faces. The algorithm used in the GPU version was applied slightly different and performs in $O(n^2)$ of complexity. For each face, it iterates through all faces again and search for faces that share the same two cells. It is not necessary to iterate two times over both faces and boundary faces arrays. As the cell structure already stores the positions of the faces and boundary faces for each cell, it is only necessary to iterate one time the cells structure and for each one, color its faces and boundary faces, reducing the complexity from $O(n^n)$ to only $O(n)$. With this scheme, when iterating the faces and boundary faces arrays for each color, all the threads will safely write in independent positions. However, this option makes the improved code execute more functions/tasks than the initial version. For each iteration over the faces and boundary faces of the initial code, the improved one must perform n more tasks, where n is the maximum number of colors.

This algorithm has not the objective of coloring the mesh with the minimum number of colors. This means that more colors than necessary may be used as can be seen in figure 4.2. Figure 4.2a shows an example of a 2D mesh with the minimum number of colors which is 3. However, the same mesh can be colored by the algorithm with 4 colors, depending of the order of which each cell has been colored as can be seen in figure 4.2b. As the "perfect" coloring scheme is an algorithm computationally intensive and that could take more time than the SIMPLE method itself, a more relaxed algorithm was used.

4.1.3 Multi-thread Implementation

With the coloring scheme method implemented, the parallelization of the code was performed by using the *OpenMP* library. Every function from the SIMPLE algorithm has been parallelized with the exception of the algorithm's control flow.

As the FlowCode consists of nearly eight thousand lines of code, with the SIMPLE algorithm taking the major part, many problems appeared during the parallelization of the CPU



(a) With 3 colors (b) With 4 colors

Figure 4.2: Example of coloring scheme on a 2D mesh

code, mostly for locating which functions generated the wrong results when parallelizing.

4.2 FlowCode MIC

Intel states [Reinders, 2012; Jeffers and Reinders, 2013] that the best way to obtain performance from the Xeon Phi coprocessor, is to improve the code on typical Intel CPUs. As this has already been done, focus has been made in the implementation of vectorization of some parts of the code. However, as the major portion of the code depends on the access of the mesh elements (which is irregular due to the use of unstructured meshes), this was not possible to implement - only small regular portions such as some updates to the mesh were vectorized. Some effort was done in compile testing by finding the required flags the execute the application as well as installing all required components such as libraries and the input meshes. The improved CPU application was ported in the Xeon Phi natively, which means that the application runs entirely in the coprocessor.

4.3 FlowCode GPU

No real improvements to the FlowCode GPU version were made. This version helped the parallelization of the CPU code. However, several performance problems have been identified, which may serve as future reference for performing improvements of this version.

The coloring scheme of this implementation performed in complexity $O(n^2)$, and although this poor approach was more or less efficient for smaller meshes, it can be an important bottleneck for larger ones. Hence, the approach used for the CPU version was also used here.

According to [Farber, 2011], the SoA format performs better for GPU accelerators. The AoS format does not allow the use of memory coalesced access. This results in many load

instructions to get data available in cache for each thread. For each warp, 32 accesses may be needed instead of 1 if each thread was accessing contiguous data in memory. Hence, changing the datastructures and how the data is represented could give and important speedup. Also, exploiting the use of shared memory to reuse data could benefit some kernels, such as the ones performing the assembly of the systems of equations.

Control flow is greatly important and most of the kernels contain branch instructions that lower performance. Since each warp thread must execute the same instructions, when there is a branch instruction, each thread must execute two times (one for each branch). This may greatly compromise performance and the GPU execution could improve if all warp threads follow the same control path. These were the main problems identified. However, profiling the application could show other problems which are not visible in the application's code.

4.4 FlowCode StarPU

This implementation uses both the improved CPU and GPU versions. The analysis performed on the two frameworks in chapter 3 showed their main characteristics. The reasons to choose StarPU are related with much more features this framework provides, being more stable as well as better documentation. The fact that this framework has been improved over several years was also a key factor. On the other hand, the DICE framework cannot be installed or configured the same way as a normal application and is undergoing several changes. As StarPU provides two methods for using the framework, the low level Application Programming Interface (API) was used instead of the C extension. The main reason was that the former provided more control of each task and the documentation is more extensive. The latter has only some simple examples which do not represent the difficulty found in the FlowCode.

To improve the performance and communication between devices, the malloc function was replaced by the starpu_malloc function to allocate all datastructures of the application. This helps StarPU to achieve more performance by delegating the allocation to CUDA which will pin the corresponding allocated memory. This enables asynchronous data transfers which may overlap with computation. Pinning memory is a great strategy for CPU applications which offloads computations to the GPU, but is even more useful for StarPU which can transfer updated data without compromising computation.

4.4.1 Parallel Approach

A common way to let StarPU create several parallel tasks that can be executed at the same time is to parallelize each task into fine grained ones. This could be seen as a similar approach with the one adopted on the CPU version. Instead of having each thread executing

some portion of work, this work would be encapsulated into a task, that could be executed by any worker. However, this approach cannot be performed for some tasks of the algorithm, since multiple tasks cannot write in the same handle (such as the right-hand sides of the system of equations) at the same time. StarPU would thereby sequentially execute each task.

To somehow overcome this problem, a higher approach has been used instead of parallelizing each task. The SIMPLE algorithm already allows the execution of several tasks in parallel.

4.4.2 System of Equation with Multiple Right-hand Sides

The process of solving the velocity components (u, w and z) for the "Momentum Conservation Equations" was performed sequentially in the CPU version. The system of equation was assembled for u, solved and then updated in the mesh. This process was repeated for both w and z. However, a closer look to the SIMPLE algorithm reveals that all velocity components can be computed at the same time, i.e., the several system of equations can be seen as only one with multiple right-hand sides. Therefore, to improve this algorithm, three right-hand side ($\mathbf{b}$) as well as three solution ($\mathbf{x}$) arrays and handles were used. The system can now be assembled for the three velocity components and solved concurrently. The downside of this solution is the use of more memory which may be important when solving problems with very large meshes.

Other tasks can be executed in parallel such as updating different elements of the mesh, such as faces, boundary faces or cells properties. However, the fact that a handle represents a AoS structure which stores faces, boundary faces, and cells prevents StarPU to create parallelism. Although the "Pressure Relaxation Field", "Update Viscosity" and "Correct Face Velocities" tasks changes different values of the AoS, as they are represented by one handle, StarPU will sequentialize the execution of these three tasks. This problem could be solved using "real" StarPU tags to inform the framework that these tasks are completely independent (instead of relying on the default submission mechanism). However, this would not be the better approach, as when finishing executing these tasks, this same handle would be invalidated on all other workers, which would require a fresh copy of all data (even though only a small section of the handle data has been changed). To allow parallelism and minimize data transfers, the structure fields that need to be updated in the SIMPLE method where removed from their AoS structure and instead, simple arrays have been used (each one with a respective handle). Therefore, only the data that has really changed will be invalidated on all other workers, minimizing data transfers and maximizing parallelism. While in the CPU version every task was executed by multiple threads but each task at a time, the StarPU version executes multiple tasks at the same time (with each one being executed also in parallel). The figure 4.3 shows the diagram of task dependencies and the tasks that can be

executed concurrently. As can be seen, the number of tasks that can be executed in parallel is limited and the maximum number of tasks executed in parallel is 4. The main reason for this limited tasks parallelism is that each tasks cannot be further divided into fine grained ones. Another approach to overcome this problem would be to use replicas. With this

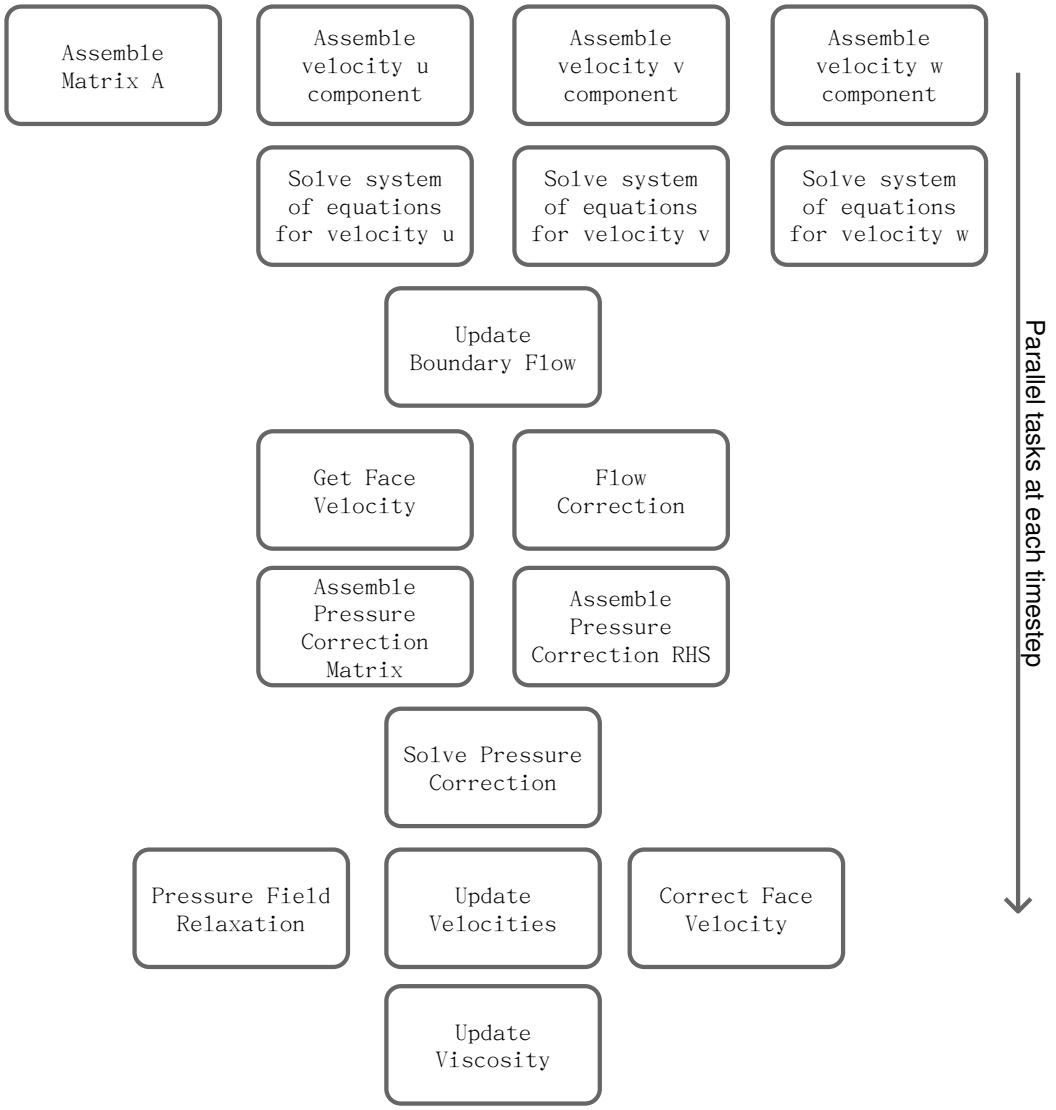


Figure 4.3: Dependencies task graph for each SIMPLE iteration

approach, several auxiliary arrays would be used per right-hand side so that each task being responsible for a number of faces (or a color) could write data into its own array. At the end of this operation, a reduction operation would be executed to assemble the data arrays to each respective right-hand side. Therefore, the three right-hand sides could be assembled at the same time, each one with multiple concurrent tasks. However, this has the cost of having several data handles over multiple arrays (as many as the level of parallelism of each assembly task) which would consume much more memory. A third approach to increase parallelism would be to remove the coloring scheme, and iterate the main structures in a cell-wise approach as stated in 4.1.2.

StarPU leverages heterogeneous platforms when there are many tasks to execute (most of them preferentially in parallel). As the SIMPLE algorithm has a stopping criteria based on the residues of the system of equations, there is no way to know the number of iterations of the algorithm. This is not a problem for applications that are not task-based such as the CPU version but may be hard to handle for the ones that are. One way to solve this problem would be to execute a bunch of tasks corresponding to one iteration and testing if the residues are lower than the predefined value. This would perform poorly since a low number of tasks would be submitted at each iteration. To overcome this problem, a simple heuristic has been used. As the user that uses the software knows and gives as input the maximum number of iterations as well the residual threshold, the StarPU version will submit one third of the iterations and perform the stopping tests after that. If all conditions are met, the program will stop. Otherwise, another one third of the maximum number of iterations will be launched. With this approach, a higher number of tasks can be submitted to StarPU without interruption. However, this means that in an example where the maximum number of iterations is 300 and the solution is found in iteration 201, the StarPU version would execute the 300 iterations of the SIMPLE method, more 99 than it was needed.

4.4.3 Challenges when using StarPU

StarPU offers several interfaces which users can use to register their data to the StarPU memory management system. The most used ones are the vector/array interface which allows to register contiguous memory positions to StarPU. StarPU also offers the possibility to create a custom interface. To try simplifying the FlowCode application, the creation of a custom interface was attempted to represent the mesh. However, due to some problems that arose when submitting tasks and which were difficult to find, this interface was dropped. The array interface was used instead to represent each array of the mesh datastructure. Also, StarPU may pose some problems for the developer, as most errors do not give enough information to find the main reasons of such errors and are not easily debugged.

Summary

This chapter presented the improvements made on the FlowCode. The original version was improved sequentially first, with techniques such as loop reordering, loop fusion or by storing auxiliary values instead of computing them in each iteration and parallelized afterwards using OpenMP. *This improved version was also ported to the Xeon Phi coprocessor in a native way. Only some improvement advices were made about the CUDA version. Both the improved CPU and CUDA versions were used to develop a heterogeneous implementa-*

tion. This has required the change of some datastructures as well as the algorithm to allow concurrent execution of the assemble functions, solver of system of equations and update functions.

Chapter 5

Profiling Results

This chapter describes the testing methodology and shows the experimental measurements. A critical analysis is made for each result, which include the sequential, OpenMP, *GPU and MIC versions. Several schedulers have been used with StarPU. Their performance over different configurations of the environment is described along with an explanation for the results obtained. This chapter ends by showing the overall speedups of all versions when compared with the single-threaded best value.*

5.1 Testing Environment

The experimental environment used to perform the measurements consists of two computational dual socket nodes from the SeARCH cluster at the Department of Informatics each with a different accelerator attached. The CPU devices at the nodes have the characteristics presented in table 5.1. The "compute-562-1" computational node contains two NVIDIA Kepler GPUs with the characteristics presented in table 5.2 and the "compute-562-6" contains two Xeon Phi coprocessors, whose characteristics are showed in table 5.3.

Model	Intel Xeon E5-2695
#CPUs	2
#Cores/CPU	12
#Thread/Core	2
Clock freq	2.40 GHz
L1 cache size/core	64 KB
L2 cache size/core	256 KB
L3 shared cache size/CPU	30 MB
RAM	64 GB (2 x 32 GB)

Table 5.1: CPU of both computational nodes

All versions of the FlowCode where compiled using gcc 4.8.2 with the exception of the MIC version. Although the gcc compiler for the Xeon Phi co-processor is supported, it lacks

Model	Tesla K20m (Kepler)
#SM	13
#CUDA cores/SM	192 (2496 total)
Clock frequency	706 MHz
L1/memory	64 KB
L2 cache	1.25 MB
Global memory	4800 MB

Table 5.2: compute-562-1 Kepler GPU

Model	Intel Xeon Phi 7120P
#CPU	1
#Cores	61
#Thread/Core	4
Clock freq	1238 MHz
L1 cache size/core	64 KB
L2 cache size/core	512 KB (30.5 MB total)
RAM	16 GB

Table 5.3: compute-562-6 Xeon Phi co-processor

some support and optimization flags that are necessary to executed code efficiently and thereby, the icc (Intel Compiler) was used instead. The GPU version was also compiled with the lasted version of CUDA (v. 6.5). The latest StarPU version has been used, currently 1.1.3 along with "hwloc" 1.9 which is used by StarPU for topology information of the compute nodes.

5.2 Testing Methodology

As the critical path of the application was the SIMPLE method, which would take more than 90% of total execution time in the sequential application, only this portion was improved. Hence, only this method was measured to obtain the speedup of each version over the sequential one. The only exception is the coloring scheme function which was also taken into account since it is needed for the parallel versions. Operations such as loading the mesh and perform initial arithmetic for the geometry were not taken into account. Each measurement was taken eight times, and the fastest three picked. If the second and third values were between the interval of 95% and 105% of the best value, this best value was picked. Otherwise, more measurements have been taken until this condition was met.

Three data sets have been used to perform the performance evaluation. These represent three common problems which are the poiseuille flow between parallel plates, lid-driven cavity and the flow around a cylinder. The number of elements of their meshes are presented in table 5.4 whereas their geometry are presented with less elements on the appendix A. Some of the meshes are refined in specific locations to improve the results' accuracy.

Input	Vertices	Faces	Boundary faces	Cells
poiseuille	3956050	3941384	3956048	1973136
ldc	1054152	1049800	1054150	525625
fac	868358	1294462	1732676	864320

Table 5.4: Input datasets

Although the CPU version of the FlowCode is able to solve flow distribution (velocity/-pressure) and heat problems, the provided GPU version does not. Hence, the measurements presented here are only relative to the first type of problems.

5.3 Performance Results without a Framework

5.3.1 Original Sequential *vs* Improved Single Threaded

As the sequential version has also been improved (before paralelization), both original and improved versions have been compared. Figure 5.1 shows the speedup obtained with the parallel version running with only one OpenMP thread ("cpu_1") when compared with the original sequential application. For the "poiseuille" and "ldc" problems, the speedups are

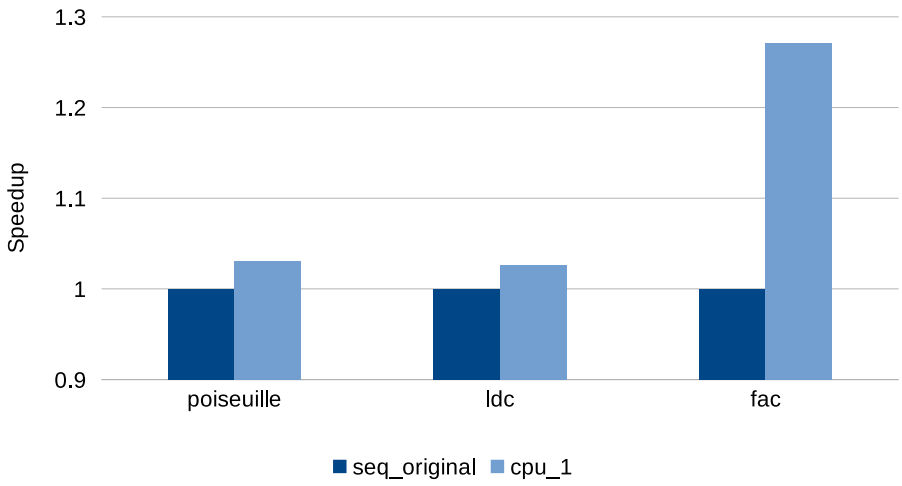


Figure 5.1: Speedups over seq_original

almost non-existent whereas with the "fac" problem, speedups almost attain 1.3. Even though the used coloring scheme implies more function calls, the techniques applied in section 4.1 surpasses this overhead. However, the sequential speedups obtained are theoretically inferior to the real speedups since there is always some overhead from the OpenMP library.

5.3.2 CPU Scalability

For the parallel version, the *OpenMP* threads were binded to their respective cores. This avoid situations where context switching of a thread from a core to another one increases cache misses due to the fact that the cache does not contain the last values (because it contains values from other thread that was previously executing on that core). Thread binding ensures that each thread will run in the same CPU core from the beginning to the end of program execution. The scalability of the parallel version when using different number of cores is presented in figure 5.2. Although it does not scale linearly, it presents reasonable speedups when using 6 cores (almost 4). However, this does not hold true for 12 and 18 cores where the speedup increases at a slower rate. For "poiseuille" and "ldc" problems, speedup starts to decrease when using hyperthreads which is the opposite of the "fac" problem that seems to benefit from them. These values reflect the behavior of the application which is

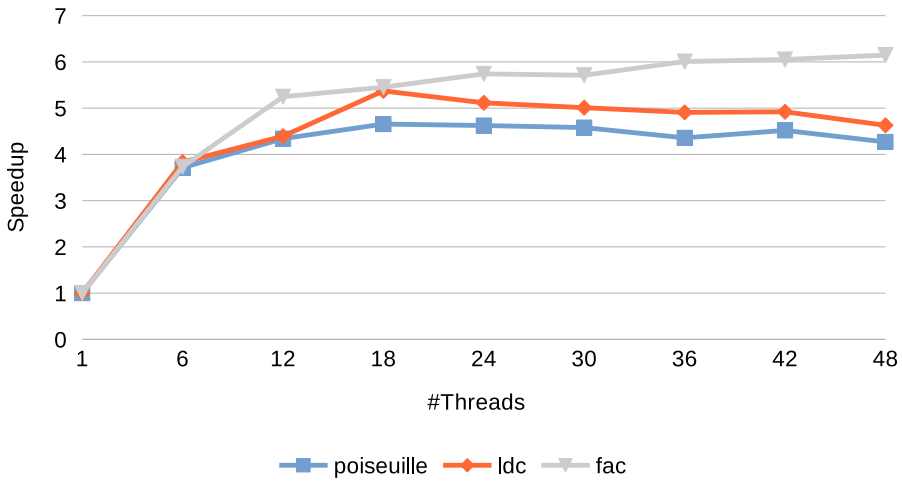


Figure 5.2: CPU scalability with increasing #threads

memory bounded - the indirections when accessing mesh values increases the cache misses significantly. The maximum speedups obtained are 4.7, 5.4 and 6.2 for the "poiseuille", "ldc" and "fac" problems respectively. This poor thread scalability suggests that a mix of processes and threads could produce mode efficient results. However, this approach would require to split the mesh so that each process execute in its own partial mesh.

5.3.3 MIC Scalability

The threads were binded in the same way as the OpenMP version to avoid cache misses. As Intel states [Reinders, 2012; Jeffers and Reinders, 2013], when improving performance on a Xeon processor, the MIC version will also benefit from these improvements. As the parallel version has been optimized, running the application natively should be enough to

have significant gains over a typical CPU. The execution was native - the application executes completely in the coprocessor - but only measurements of the SIMPLE (and coloring scheme) algorithm were taken (which are the paralelized functions). Offloading the SIMPLE algorithm only, instead of a native execution would not really improve global performance. Figure 5.3 shows the scalability of the MIC version when running with different number of threads and using 60 cores. The last core has not been used because the co-processor runs a OS inside which uses one core and as stated in [Rahman, 2013], "often a 61 core processor may end up with 60 cores available for pure computation". The results show that the MIC

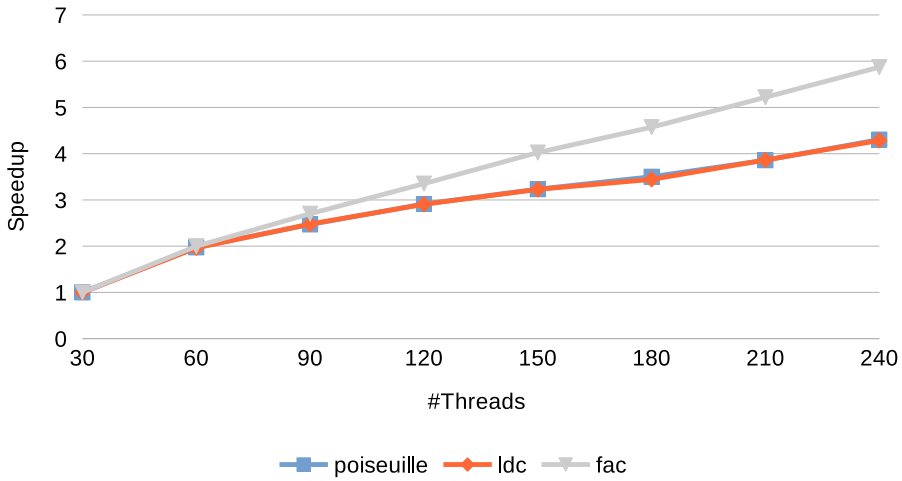


Figure 5.3: Xeon Phi scalability with increasing #threads

version has a better scalability than the CPU version. From 30 to 60 threads, the improvements are linear. When using hyperthreads, the improvements are slightly less significant. As explained in 3.2.2, the maximum device performance can be obtained theoretically when using 120 threads (when an instruction may be issued by each core in each clock cycle), if the program and the execution are perfect. Hence, it is interesting to note that when using 180 and 240 threads, the performance always increases meaning that adding more threads help lowering memory indirections problems and leverage the memory bandwidth of the Xeon Phi.

5.3.4 CPU *vs* GPU *vs* MIC

The GPU version has not been improved apart from the coloring scheme but other things have been changed, such as function calls to the [NVIDIA, 2013b] library, since the API has changed for some functions from CUDA 5 to CUDA 6.5. All versions speedup were measured against the single-threaded improved version. Figure 5.4 shows that the GPU version is the slower one for all inputs. This may seem strange, since a GPU is typically faster than a CPU, but this dissertation has not the objective to really compare the devices but the performance

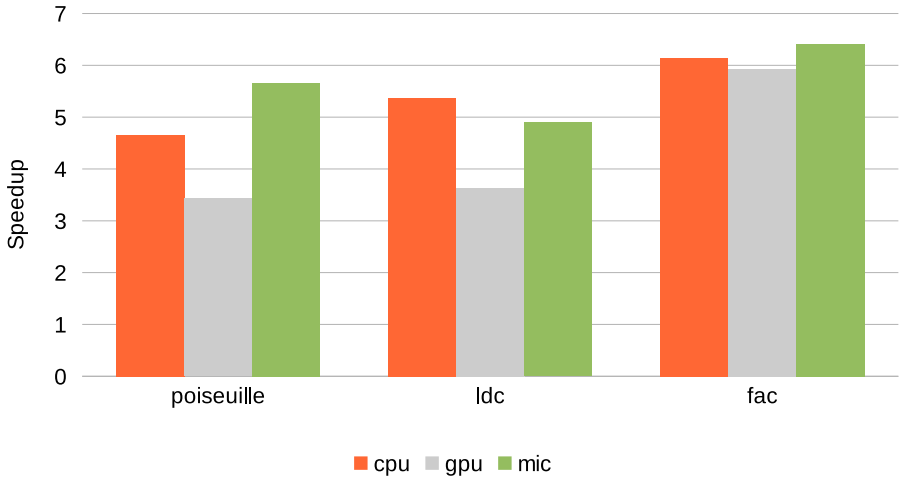


Figure 5.4: Speedups over cpu_1

obtained for each version instead. Since the GPU version has not been improved, it would not be fair to perform an evaluation of the devices performance. The MIC version achieves the best results for the "poiseuille" and "fac" problems and it is only surpassed by the CPU version for the "ldc" problem. The maximum speedup obtained by the best version is 5.7, 5.4 and 6.4 for the "poiseuille", "ldc" and "fac" problems respectively.

5.4 Performance Results with StarPU

As StarPU provides several schedulers with different characteristics, which have impact on the execution time, five of them have been chosen, which are "dm", "dmda", "ws", "pheft" e "peager". To further understand the impact of CPU and GPU devices for each scheduler of the framework, measurements were taken for different configurations of the testing environment - CPU only, 2 GPUs, CPU + 1 GPU and CPU + 2 GPUs. The MIC coprocessor was not used with StarPU. Although there is some support, it only exists in the trunk repository and it has not been released officially. Some effort has been made to use it, but compile problems appeared which have not been solved by the StarPU team. This prevented performing measurements on a distinct platform, which would emphasize the importance of heterogeneous frameworks to develop applications that can execute in platforms with different devices.

The framework has an option to show the number of tasks executed in each device, which helps to explain the performance results of each scheduler. However, these results have not been taken into account since this option affects (although only slightly) the measurements.

5.4.1 Performance with CPU

Figure 5.5 shows the performance of the five schedulers when executing only in the dual socket CPU when compared with the parallel OpenMP version. The parallel versions of the "eager" and "heft" schedulers are the faster ones. These results were expected, since the two can execute each task across all CPU cores. However, "pheft" presents better results. By analyzing the number of tasks executed by both schedulers, it seems that "pheft" executes more tasks on more cores than "peager", where a fewer number of cores are responsible for executing most of the tasks. Nevertheless, their performance results are worst than the best OpenMP result, mostly due to the StarPU scheduling overhead. Also, as the task

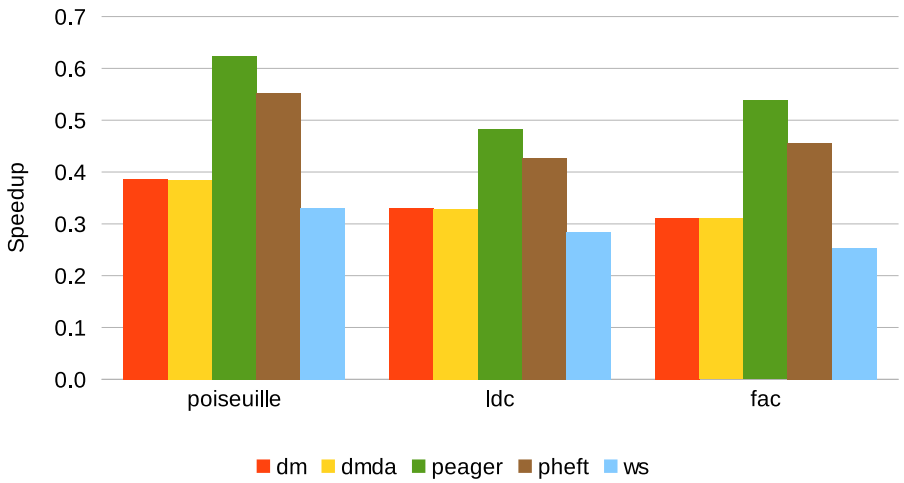


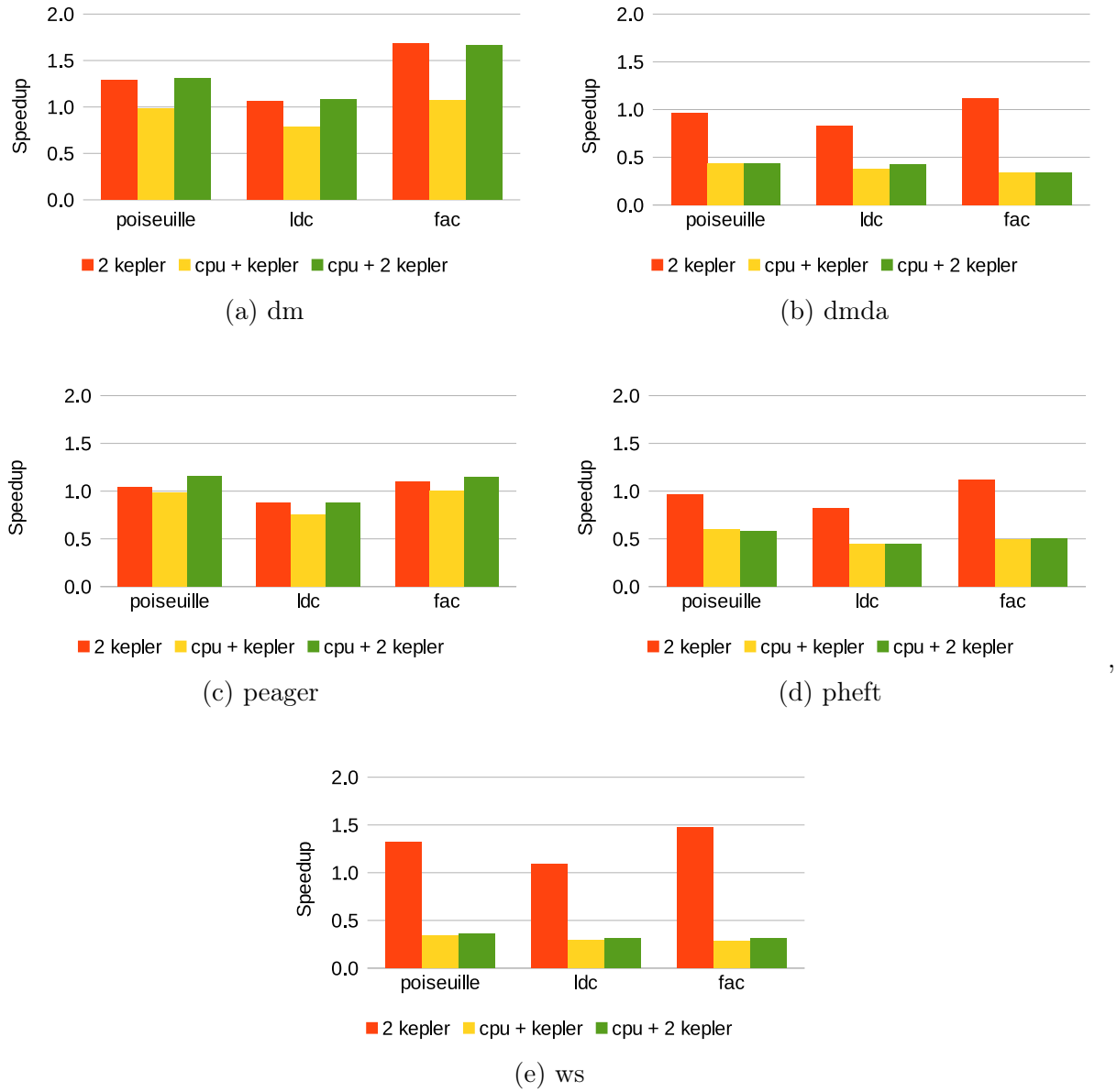
Figure 5.5: StarPU with CPU only

parallelism is limited (as showed in figure 4.3), the other schedulers do not leverage all CPU cores at the same time. Instead, only a maximum of 4 cores can execute tasks simultaneously. The "dm" and "dmda" schedulers have similar results which are explained by the fact that both are the same scheduler when there is no data awareness, i.e, when there is no need for data transfers between devices (which is the case). The "ws" is the slowest scheduler due to the fact that each core will pop out a task from the queue in order, providing a balanced execution over all CPU cores. With this scheduling policy, the core caches will not have the values needed when starting executing a task and a high number of cache misses will occur. This is the main difference when compared with the previous two schedulers (which mostly use the same cores during all execution).

5.4.2 Performance with CPU and GPUs

The last section presented values suggesting that it does not make sense to use StarPU to parallelize an application only in a (dual socket) CPU if it will use OpenMP to do that. StarPU only adds overhead over an application which uses OpenMP. This is different when

using more than one device of different types. Figure 5.6 shows the performance obtained by



(a) dm

(b) dmda

(c) peager

(d) pheft

(e) ws

Figure 5.6: Schedulers performance over different system configurations

the schedulers over different configurations of the testing environment when compared with the parallel OpenMP version. The "dm" scheduler has speedups when using it with 2 GPUs or 1 CPU and 2 GPUs. For the "fac" problem, both almost attain 1.7 of speedup, which are good results specially when using two GPUs, since adding the CPU does not improve the result. Using the CPU and only a GPU does not provide good results since a task on the CPU will only execute on one core and there are only a maximum of 4 tasks (as showed in figure 4.3) than may execute concurrently (1 on the GPU and 3 on the CPU) and thereby, synchronization overhead between the devices penalizes the lack of computation.

The "dmda" shows the worst results of all schedulers. Analysis of the number of tasks

executed in each device over the different configurations shows that the scheduler tends to execute most of the tasks on the CPU (which again, only executes a task on one core). The objective of the scheduler is to minimize the execution time of each task alone and not all tasks. Since the scheduler is data aware, the time spent transferring the data needed by the task and executing it is greater than executing locally (in the CPU), which is why most tasks are executed in the CPU. Again, figure 5.6 shows that there are several sets of tasks that can be executed serially but are independent of each other. This means that executing this sets on different devices could improve performance but "dmda" cannot schedule the tasks in this way since it is data aware. The best results are obtained when using only two GPUs, but again, only one GPU executes most of the tasks. Therefore, the poor results are consequence of low task parallelism (which affect mostly the CPU which has many cores) and little use of the devices because of the data awareness.

The "peager" scheduler does not suffer from low parallelism on the CPU device since it can use OpenMP to execute a task using multiple cores. It suffers however from not been able to prefetch data before the tasks needs it, since the scheduling decisions are taken late. Results show that the maximum speedup obtained is of almost 1.2 for the "fac" problem when using all resources of the compute node. Although the scheduler does not give the best results when comparing with other ones, it seems to behave steadily independently of the configuration.

As the "pheft" is similar to the "dmda" scheduler, the results are more or less the same, with the difference that when using a configuration that uses the CPU, the performance achieved is greater. This indicates that it does not make sense to use the "eager" and "dmda" schedulers, since the "peager" and "pheft" schedulers have the same characteristics and also provide parallel tasks.

For the last scheduler, "ws", it only offers good performance when using two GPUs. When using GPUs and CPU cores, each worker executes a similar amount of tasks, which will penalize the configurations that uses the CPU due to the lack of parallel tasks. A speedup of 1.32 and 1.47 is obtained for the "poiseuille" and "fac" problems respectively.

5.4.3 Overall Speedup

The figure 5.7 shows the speedups obtained from the several versions developed when compared with the single threaded improved. For the StarPU version, only the best value from all schedulers and system configurations has been taken. The heterogeneous version is the faster one for each input problem and attains a speedup of 6.2, 5.9 and 10.35 for the "poiseuille", "ldc" and "fac" problems respectively.

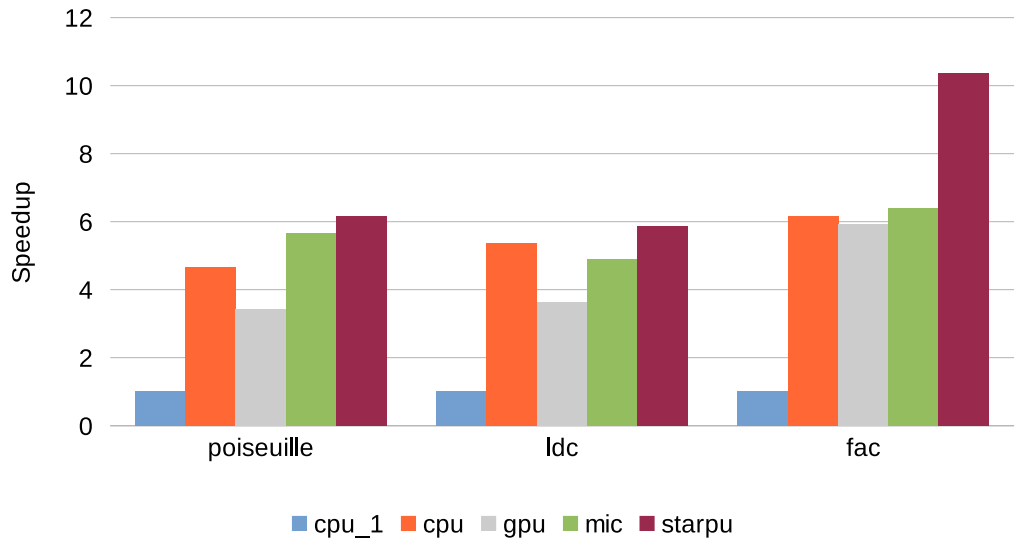


Figure 5.7: Overall speedup

Summary

This chapter showed that the improved CPU version presents better results for all inputs when compared with the GPU version and that it is possible to obtain good results on the Xeon Phi accelerator only by improving it on a Xeon CPU. According to the results, StarPU reveals to leverage the computational power of a heterogeneous platform, improving the GPU version by using two GPU devices, even tough, the paralelization is limited by the number of tasks that can be executed at the same time. The same also happens with some schedulers when using all resources of the testing environment (although the results are less expressive). Several conclusions can be made about the schedulers used. The performance of each scheduler depend on the characteristics of the application as well as the characteristics of the system were the application runs. Some schedulers are useless as there are other ones that are similar and behave better for CPU since they can execute parallel tasks.

Chapter 6

Conclusions and Further Work

6.1 Conclusions

Polymer processing is a process that has on the laboratorial experimentation and calibration its downsides, as they represent high production costs. However, software applications have been developed to model and simulate these tasks. The FlowCode application, which aids the design of extrusion forming tools, is one example. Two versions of this code have been developed, one to be executed on GPU devices as computing accelerators, another for sequential execution on a single CPU core. In an era where multiple devices may co-exist in a single system, each one with its characteristics, it is crucial that these applications fully leverage the available computing power of heterogeneous systems. Most heterogeneous systems available nowadays contain CPUs and accelerator devices such as *NVIDIA* GPUs and Intel Xeon Phi's. As a GPU version had already been developed, the focus of this work was on improving and parallelizing the CPU version.

Profiling the application identified the most computationally intensive as being a function in the SIMPLE method, which is the main method of the FlowCode to solve several systems of equations in each time-step. Therefore, this function was the main target of improvements in the sequential application. Using several techniques, the improved version is now able to run faster than the original single core code, even though the coloring scheme implemented for the paralelization reduces some of the benefits. Using the multiple cores available in the testing environment, we managed to achieve speedups between 4.5 and 6, which provides better results than the GPU implementation. This improved version has also been ported to the MIC accelerator, a many-core device with 61 Pentium cores, where it was run natively. Results shows that this version runs faster in each case, when comparing to a dual 12-core CPU device and GPU implementations. Speedups on the MIC ranges from 4.9 and 6.4, when comparing with the improved single threaded version.

With all implementations developed, focus have been made into heterogeneous platforms. The StarPU and DICE frameworks were studied, and their main strengths and weaknesses

identified. With this evaluation, the StarPU framework has been chosen to develop an heterogeneous application and to control its execution, with adequate work balance and data transfers. Several techniques have been implemented to allow the execution of tasks on several devices at the same time, even though the parallelism is limited due to some constraints. The coloring scheme prevents parallelism when gathering all components for the system of equations and the solution of each system is done entirely on one worker. Although some of these tasks can be concurrently executed due to the nature of the SIMPLE method, each one cannot be further split into tasks with a finer grain. The measurements were taken using five different schedulers provided by StarPU, with different execution configurations across the devices of the environment. When executing the application only on the CPU, the StarPU version (with the five schedulers) is slower than the *OpenMP* version, as expected since the CPU parallel version already uses all cores of the system and StarPU only creates overhead. When executing on the two GPUs, a maximum of 1.7x and 1.5x speedup has been obtained with the "dm" and "ws" schedulers, respectively when comparing with the improved single-core CPU version.

Results when executing the application on both CPU and GPUs are slower than using only the two GPUs, with most schedulers. This results can be explained by the lack of parallelism and for most schedulers since that each core is a single worker (as with the GPUs). This means that for twelve worker configuration (10 cores + 2 GPUs), only 4 may execute at each time. Global evaluation of the results shows that the StarPU version is the faster one with speedups varying between 5.9, and 10.35, when comparing with the improved single threaded application. Following are the MIC and CPU versions. The GPU version is the slower one, but this implementation is far from being efficient and it has not been improved throughout this dissertation. Therefore, any comparison between devices such as MIC and GPU would not be fair.

The results obtained, although limited by the parallelism of the application and the schedulers of the framework, shows the benefit of heterogeneous platforms and frameworks to leverage the available computing power of many different computing systems with a single application.

6.2 Further Work

Several techniques have been discussed throughout the dissertation which have not been tested. Some of these are different approaches to solve the same problems, and may or may not improve the applications' performance. For the StarPU application, the parallelism was limited and was done at a higher level. It would be interesting to follow a cell-wise approach instead of a coloring scheme to allow the parallelism of each tasks into fine grained ones. This approach would create multiple levels of parallelism, generating more independent tasks

and benefiting CPU sequential schedulers such as “dm” and “dmda” as they do not support openMP. This approach would perhaps benefit more the DICE framework since it can find the best granularity for each task. Therefore, further work should also be focused on the DICE framework to perform a fair comparison between them.

Testing the framework with the Xeon Phi co-processor was not possible due to unsolved bugs on the StarPU application in the trunk repository. But once these problem are solved, obtaining results from the framework is an easy task since there is nothing to do to other than compile the application with MIC support. Therefore, it would be interesting to compare the performance obtained with a Xeon Phi without the framework and two Xeon Phi’s and see if the improvements are on the same order with the ones obtained when using GPUs.

This dissertation focused on the CPU, MIC and StarPU versions. But, as stated before, several improvements may be done in the GPU version such as focusing in vectorization, by changing the datastructrures format from AoS to SoA. These improvements would also benefit the applications which use the frameworks.

Bibliography

E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, Sept. 2010.

C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, Aug. 2009. Springer.

C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*, Shanghai, China, Dec. 2010a.

C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Rapport de recherche RR-7240, INRIA, Mar. 2010b.

C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.

J. Barbosa. GAMA Framework: Hardware Aware Scheduling in Heterogeneous Environments. Technical report, ICES, University of Texas at Austin, September 2012.

J. Barbosa, S. Keely, R. Ribeiro, D. S. Fussel, C. Lin, A. Proença, and L. P. Santos. Automatic Granularity Control for Irregular Workloads on Heterogeneous Systems. Submitted to the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'2015.

R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

O. S. Carneiro and J. M. Nóbrega. *Design of Extrusion Forming Tools*. Smithers Rapra Technology, 2012.

L. Courtès. C Language Extensions for Hybrid CPU/GPU Programming with StarPU. Research Report RR-8278, INRIA, Apr. 2013.

R. Farber. *CUDA Application Design and Development*. Morgan Kaufmann, 2011.

C. Geuzaine and J.-F. Remacle. Gmsh: a Three-Dimensional Finite Element Mesh Generator with built-in pre- and post-Processing Facilities. *International Journal for Numerical Methods in Engineering*, pages 1309–1331, 2009.

N. D. Gonçalves, O. S. Carneiro, and J. M. Nóbrega. Design of Complex Profile Extrusion Dies Through Numerical Modeling. *Journal of Non-Newtonian Fluid Mechanics, Volume 200*, pages 103–110, Oct. 2013.

N. D. F. Gonçalves. *Computer Aided Design of Extrusion Forming Tools for Complex Geometry Profiles*. PhD thesis, University of Minho, 2014.

A. Hugo, A. Guermouche, R. Namyst, and P.-A. Wacrenier. Composing Multiple StarPU Applications over Heterogeneous Machines: a Supervised Approach. In *Third International Workshop on Accelerators and Hybrid Exascale Systems*, Boston, USA, May 2013.

INRIA. *StarPU Handbook: for StarPU 1.1.3*. INRIA, 2014.

Intel. Using Intel® Math Kernel Library for Matrix Multiplication (C Language), 2013a.

Intel. Intel® Math Kernel Library for Linux* OS: User's Guide, 2013b.

Intel. Intel Math Kernel Library: Reference Manual, 2013c.

J. Jeffers and J. Reinders. *Intel® Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann, 2013.

I. Kampolis, X. Trompoukis, V. Asouti, and K. Giannakoglou. *CFD-based Analysis and two-level Aerodynamic Optimization on Graphics Processing Units*, page 712–722. Elsevier, 2009.

NVIDIA. *NVIDIA Kepler GK110 Next-Generation CUDA Compute Architecture*. NVIDIA, 2012a.

NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 (Whitepaper)*. NVIDIA, 2012b.

NVIDIA. *CUDA C Programming Guide*. NVIDIA, July 2013a.

NVIDIA. *CUDA CUSPARSE Library*. NVIDIA, July 2013b.

OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 4.0 edition, July 2013.

R. Rahman. Intel® xeon phi™ core micro-architecture. 2013.

J. Reinders. An Overview of Programming for Intel Xeon Processors and Intel Xeon Phi Coprocessors. 2012.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. ISBN 0898715342.

Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986. ISSN 0196-5204.

Appendix A

Input meshes

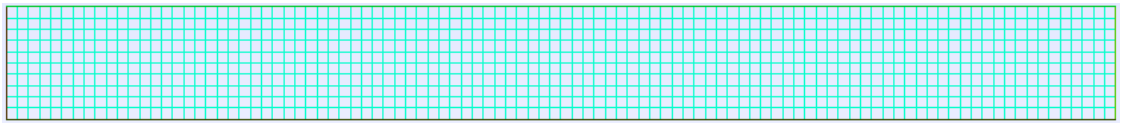


Figure A.1: Poiseuille

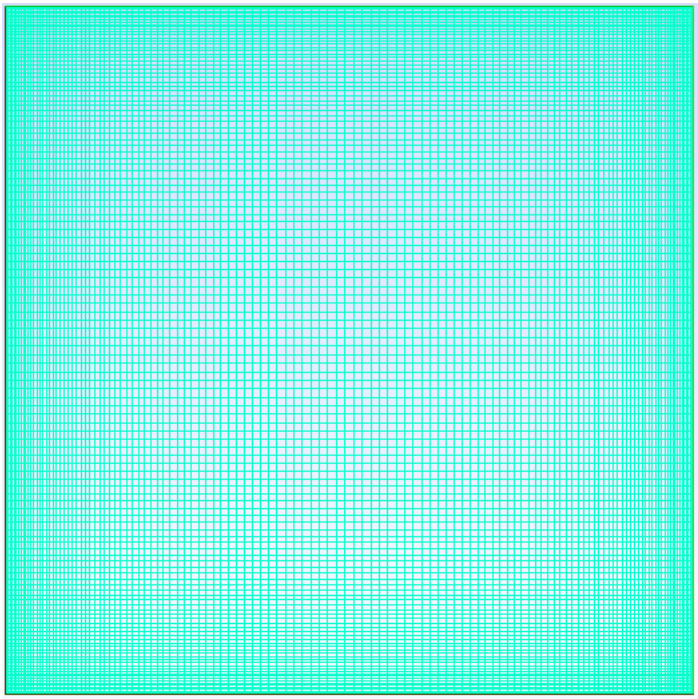


Figure A.2: Lid-driven cavity

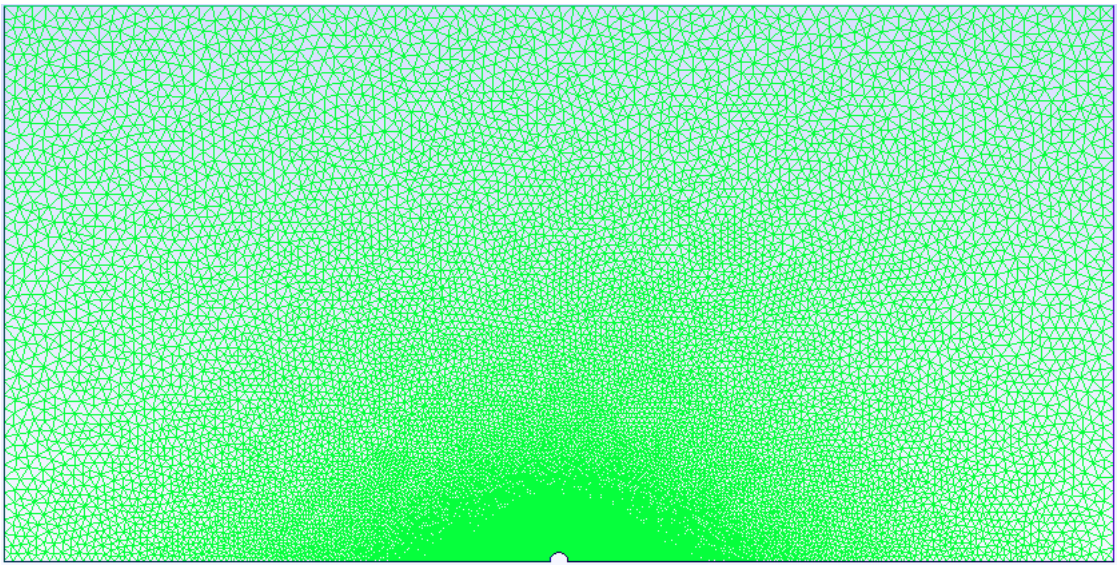


Figure A.3: Flow around a cylinder