# Design and Optimization of OpenFOAM-based CFD Applications for Modern Hybrid and Heterogeneous HPC Platforms

Thesis by

**Amani AlOnazi**

In Partial Fulfillment of the Requirements

For the Degree of

**Master of Science**

King Abdullah University of Science and Technology, Thuwal,

Kingdom of Saudi Arabia

February, 2014

The thesis of Amani AlOnazi is approved by the examination committee

# ABSTRACT

Design and Optimization of OpenFOAM-based CFD Applications for Modern Hybrid and Heterogeneous HPC Platforms

Amani AlOnazi

The progress of high performance computing platforms is dramatic, and most of the simulations carried out on these platforms result in improvements on one level, yet expose shortcomings of current CFD packages. Therefore, hardware-aware design and optimizations are crucial towards exploiting modern computing resources. This thesis proposes optimizations aimed at accelerating numerical simulations, which are illustrated in OpenFOAM solvers. A hybrid MPI and GPGPU parallel conjugate gradient linear solver has been designed and implemented to solve the sparse linear algebraic kernel that derives from two CFD solver: icoFoam, which is an incompressible flow solver, and laplacianFoam, which solves the Poisson equation, for e.g., thermal diffusion. A load-balancing step is applied using heterogeneous decomposition, which decomposes the computations taking into account the performance of each computing device and seeking to minimize communication. In addition, we implemented the recently developed pipeline conjugate gradient as an algorithmic improvement, and parallelized it using MPI, GPGPU, and a hybrid technique. While many questions of ultimately attainable per node performance and multi-node scaling remain, the experimental results show that the hybrid implementation of both solvers significantly outperforms state-of-the-art implementations of a widely used open source package.

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisors Alexey Lastovetsky and David E. Keyes for the continuous support of my study and research, for their patience, motivation, enthusiasm, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. It has been an honor to work with you.

My sincere thanks also goes to both King Abdullah University of Science and Technology (KAUST) and University College Dublin (UCD) for exciting and encouraging environments in which to fulfill and complete my research. I would like to thank the Erasmus Mundus Gulf Countries program for giving me the opportunity to study in Dublin. Special thanks goes to both Heterogeneous Computing Laboratory group at UCD and Strategic Initiative in Extreme Computing at KAUST for their support and help. I really appreciate all your guidance. Also, I would like to express my appreciation to the Computational Mechanics Group from the School of Mechanical and Materials Engineering at UCD for their help.

The most special thanks goes to my family especially my parents for their unconditional love and support throughout my life.

# TABLE OF CONTENTS

# LIST OF ABBREVIATIONS

| | |
|---|---|
| CFD | Computational Fluid Dynamics |
| MPI | Message Passing Interface |
| OpenFOAM | Open source Field Operation And Manipulation |
| PDE | Partial Differential Equations |
| PISO | Pressure Implicit with Splitting of Operators |

# LIST OF SYMBOLS

| | |
|---|---|
| $p$ | Pressure Field |
| $u$ | Velocity Field |
| $\nu$ | Kinematic Viscosity |

# LIST OF FIGURES

# LIST OF TABLES

# Chapter 1

# Introduction

Computational Fluid Dynamics (CFD) is a cornerstone in the understanding of many scientific and technological areas such as meteorological phenomena, aerodynamics, and environmental hazards. Computational science enabled by High Performance Computing (HPC) makes it possible for scientists and researchers to simulate these phenomena in a virtual laboratory. CFD is extensively used throughout the whole process, from early concept phases to the detailed analysis of a final product. However, CFD packages, until recently, have been aimed at homogenous systems and the parallel approach is mostly based on the process-level Message Passing Interface (MPI).

On the other hand, the progress of HPC platforms over the recent years has been dramatic, reaching the Petascale computing level and aiming towards the Exascale. Moreover, the underlying architecture is no longer an interconnected set of homogenous uni-processors each with its own memory system. It is growing more complex, hybrid, and heterogeneous. A typical compute node on modern HPC platforms comprises multi-core processors and graphic unit devices (GPUs) that act as co-processors or accelerators. This trend in the HPC platforms invites redesign of the CFD packages or the algorithms themselves to use these platforms efficiently. Therefore, there is an increasing demand to optimize the current CFD packages aimed at better utilization of the modern computing resources.

Open source Field Operation And Manipulation (OpenFOAM) [1] is a library written in C++ used to solve partial differential equations (PDEs). It consists of a large set of pre-processing utilities, PDE solvers, and post-processing tools. An attraction of OpenFOAM is in its modularity, which leads to an efficient and flexible design. It features a wide range of solvers employed in CFD, such as Laplace and Poisson equations, incompressible flow, multiphase flow, and user defined models [2]. OpenFOAM uses MPI to provide parallel multi-processors functionality, which scales well on homogenous systems but does not fully utilize potential per-node performance on hybrid heterogeneous platforms.

Current high performance computing systems are complex and difficult to utilize and manage at their extremes. The heterogeneity of these platforms leads to several challenges and much contemporary attentions devoted to new software solutions. Indeed, there is an increasing interest in implementing and optimizing applications for hybrid parallel systems, which employ different parallelization levels. Two of the main challenges are data distribution and workload balancing across the heterogeneous devices.

In this work, we selected the OpenFOAM CFD package because of its popularity as an open source library with broad adoption. We specifically targeted two OpenFOAM solvers: icoFoam, which is an incompressible flow solver, and laplacianFoam, which solves the Laplace equation. We studied and optimized the most expensive components of the selected solvers. According to the profiling results and the performance analysis of icoFoam and laplacianFoam, most of their execution time is spent in sparse linear algebraic solver, which is, in our case, the conjugate gradient solver.

Therefore, improving the linear solver will provide significant improvements to the whole application.

A set of optimizations has been introduced, which are aimed at accelerating the execution time of icoFoam and laplacianFoam solvers on modern heterogeneous nodes beyond their current performance. A hybrid conjugate gradient solver has been designed and implemented combining MPI and CUDA routines. Consequently, a heterogeneous decomposition method has been designed and implemented to decompose the computational domain across heterogeneous devices and aimed at balancing the workload of the devices during the execution of the whole applications. This method combines the traditional domain decomposition methods, which are supported by OpenFOAM, and the state-of-the-art heterogeneous data partitioning algorithms. Moreover, algorithmic improvements of the conjugate gradient linear solver that are inspired by the recent work of Ghysels and Vanroose [3] have been implemented.

The implementation of the project is broken into the following stages:

- Analyze the structure and parallel approach of the OpenFOAM code.
- Study the mathematical models of the solvers icoFoam and laplacianFoam.
- Benchmark and profile both selected solvers in order to identify the most time consuming kernel. The profiling result reveals that the conjugate gradient linear solver in icoFoam and in laplacianFoam is the most time consuming kernel.
- Study the conjugate gradient linear solver both in OpenFOAM and the Cufflink library [4], which extends the linear solver capability to run on GPU devices using the CUSP library [5].
- Evaluate the performance gain of the optimizations on the execution time of the selected solvers.

The contributions of this thesis are as follows:

- We design and implement a hybrid conjugate gradient linear solver for execution on multi-core/multi-GPU platforms and integrate it into OpenFOAM.

- We propose and demonstrate a heterogeneous decomposition method aimed at balancing the load of heterogeneous computing devices in modern multi-core/multi-GPU platforms. This method combines the domain decomposition methods in OpenFOAM and the-state-of-the-art heterogeneous data partitioning algorithms.

- We implement a recently introduced pipeline conjugate gradient algorithm [3] and parallelize it using GPGPU, MPI, and hybrid MPI+GPGPU. Our implementations of the pipeline conjugate gradient are integrated into OpenFOAM.

- We deploy and evaluate the solvers on two hybrid and heterogeneous platforms.

This thesis is organized as follows. Chapter 2 presents background research relating to OpenFOAM CFD package and heterogenous computing. In Chapter 3, we study and analyze the parallel and sequential run of the selected solvers. We present the implementation and design detail of the hybrid approach and the heterogeneous decomposition in Chapter 4. Chapter 5 compares and evaluates the performance of our optimizations using a wide array of problem sizes and test cases. Chapter 6 discusses future work and concludes the thesis.

# Chapter 2

# Background

## 2.1 Related Work

In the past decade, a number of parallel packages designed for CFD have been proposed such as OpenFOAM [1], PETSc [6], Sierra [7], and Deal.II [8]. These packages are aimed at homogenous systems, use MPI-based bulk synchronous processing parallelization, and pay no special attention to the underlying hardware. The addition of GPUs to high-end scientific computer systems anticipates new achievable performance levels. The world's most powerful platform is Titan at OakRidge National Laboratory, which is a hybrid architecture that combines CPUs and GPUs [9]. Significant work has been done on GPUs, most of it for linear algebra routines such as QR factorization on multi-GPU [10], Cholesky factorization [11], sparse direct solvers [12], Conjugate Gradient and multi-grid solvers [13, 14, 15, 16, 17]. While linear system solution generally leads the implementation wave onto new hardware, because it is a key kernel, there is new significant interest in migrating whole applications to the hybrid environment.

As a result, several works attempt to employ GPGPUs in CFD codes and much more attention is focused on sparse linear algebra kernels, such as implementing the conjugate gradient on GPU [18, 14, 19]. These works include the Cufflink library

[4], which extends the OpenFOAM linear solvers capabilities to perform on GPUs. However, no research so far in the simulation area appears to have shipped an entire CFD computations onto heterogeneous hybrid architecture. The major problem is that most CFD packages have a large source code base. Additionally, algorithmic effects of hybrid execution of parallel CFD computations, including their impact on accuracy and convergence tradeoffs in iterative methods, are not well studied yet.

Heterogeneity is one of the most challenging features of modern HPC systems. Modern CPUs consist of multi-core processors because concurrency is the remaining means of exploiting Moore's Low. Moreover, several multi-core processors are connected to a shared main memory, forming a tightly coupled multiprocessor system often called Symmetric Multi-Processing (SMP). Furthermore, a SMP can have access to special purpose processors often called co-processors such as GPU devices. As mentioned above, GPU provides potential compute performance that has crafted computational inroads against modern CPUs. This trend in HPC platforms leads to several challenges and novel approaches. Indeed, HPC programming models have not seen such innovation for more than 25 years.

Based on early pioneering works [20, 21], a tool [22] has been designed that ports the scalable linear algebra package (ScaLAPACK) [23] to the Heterogeneous Message Passing Interface (HeteroMPI) [24] aimed at heterogeneous networks of computers. This work presents algorithms that are mainly aimed at distributing and mapping the data of linear algebra applications to the heterogenous environment. A recently published work [25] claims an optimal partitioning shape for parallel matrix-matrix multiplication with heterogeneous processors.

These heterogeneous algorithms have been designed based on a performance model representing the target platform by a vector of positive numbers whose values represents the speed of the associated processing elements. This performance model is used to partition the matrix into certain number of sub-matrices and each portion contains number of elements that approximately is proportional to the speed of the associated processor. This model is often called the constant performance model [26].

More advanced models have been proposed, which take into account the memory hierarchy, such as the functional performance model [27, 28, 29]. The constant performance model assumes the data fully fits the main memory. Whereas, the functional performance model [30, 31, 32, 33] takes into account the possible presence of paging and the degradation of speed accordingly. Lastovetsky and Raddy [34] estimate the speed function of each processor for array of problem sizes during the execution of a computational kernel, which is illustrated in dense matrix-matrix multiplication.

Over recent years, the functional performance model has been adopted for the dynamic load balancing algorithms and multi-core/multi-GPU systems [35, 36, 37]. Accordingly, several partitioning algorithms have been designed and analyzed for hierarchical and heterogeneous architecture demonstrated using matrix-matrix multiplication and dense linear algebra routines [38, 39, 40].

Pioneering CFD works address heterogeneous and hybrid systems. A parallel simulation of oil extraction has been designed and optimized for heterogeneous networks of computers [41] by applying the performance model of the target platform. Gropp et al. [42] studied the hybrid MPI+OpenMP programming model on unstructured implicit CFD solvers and its affects on obtaining per-processor efficiency and parallel efficiency. Accordingly, there have been some attempts to obtain a hybrid parallelization in CFD on multi-core platforms [43, 44] such as the CFD solver TAU [45] that provides hybrid MPI+OpenMP parallelization. There are, also, matrix and graph partitioning library, which considers a heterogeneity of the devices such as METIS [46] and SCOTCH [47]. These two libraries assume a given percentage, which represents the volume of computations per each processor.

In CFD packages, there have been few attempts recently to investigate the parallel model of hybrid MPI+GPGPU. Papadrakakis [48], attempts to balance the computation across heterogeneous hybrid CPU+GPU system. The balancing is performed during the runtime by using task-based parallelism and migrations between the compute devices. On the other hand, a general framework is supported by StarPU [49, 50] project, which provides and supports task-based programming and scheduling the provided tasks on heterogeneous platform, which combines CPUs and GPUs.

The OP2 project [51] provides a framework to implement CFD applications using unstructured meshes on different computational hardware including multi-cores and GPUs (many-cores) systems. The parallelization scheme is similar to task-based parallelism in the way it handles the data dependencies. However, it is not applicable to multi-block codes as is the case in OpenFOAM and there is no special attention to the heterogeneity in the domain decomposition method, which is performed at the MPI level. Moreover, it does not discuss the multi-GPU, which is noted in the paper as a future work.

In this work, we propose a set of optimizations that combine the current HPC trend and OpenFOAM for heterogenous hybrid platforms. As discussed earlier, the current available parallelization in OpenFOAM is either multi-core/multi-processor parallelization MPI [1], or GPGPU parallelization [4]. There appears to have been no published research on developing a hybrid MPI+GPGPU solver in the OpenFOAM package, which would aim at exploitation and utilization of hybrid heterogeneous platforms.

## 2.2 OpenFOAM CFD Package

Computational Fluid Dynamics is an interdisciplinary research area that integrates physics, applied mathematics, and computer science. It can be defined as the set of methodologies and numerical methods that enable the computer to solve and provide a simulation of fluid motions. Fluid flows are encountered in many areas such as meteorological phenomena and aerodynamics [52]. Most of these phenomena are described by partial differential equations that can be discretized in the form of algebraic equations, which can be solved using computers. The CFD package on which this study is based is OpenFOAM [1]. We selected OpenFOAM because of its popularity as an open source library with broad adoption.

OpenFOAM is written in C++ and it heavily depends on C++ object oriented features such as operator overloading, inheritance, and templates. The package can be used to solve general continuum mechanics problems [2]. As with many other CFD codes, it contains three main components, namely, pre-processing utilities, Partial Differential Equations (PDE) solvers, and post-processing tools. The OpenFOAM main library provides a solution framework, including mesh handling, finite volume discretization method, linear system solvers, data structure and input and output handling. We briefly examine the task of each of these components within the context of OpenFAOM.

The preprocessing utilities handle the input of a flow problem to the CFD solver by means of a set of transformation of the provided input into a suitable form that can be used by the solver. The input of a flow problem is a specification of the computational domain, the equations to be solved in it, and its boundary and initial conditions. OpenFOAM provides a mesh generation tool called blockMesh, which generates meshes of hexahedra from a blockMeshDict configuration file. This file contains keywords and values, which are given by the user that determines the scale, shape and type of the cells within the computational domain. This step subdivides the geometry into number of smaller non-overlapping elements called grid or mesh of cells. After that, the user must specify the appropriate boundary conditions at cells, which coincide with the domain boundary [53].

The mesh is described by four basic building blocks: points (or vertices), edges, faces, cells, and boundary. A point is a vector holding the corresponding coordinates of its location. It can be a point in 3D-space or 2D-space. The points are compiled into a list, and a label, which represents its position in the list, refers to each point. A face is an ordered list of points represented by their labels. The faces also are compiled into a list, and a label refers to each face. A cell is a list of faces represented by their labels. A patch is a list of faces that clearly must contain only boundary faces and no internal faces. A boundary is a list of patches, each of which is associated with a boundary condition [2].

The numerical methods that form the basic of any solver of flow problem can be summarized in three phases [52]. The first phase is approximating the flow variable, e.g., pressure. The second phase is applying a discretization method to estimate the partial differential equations by a system of algebraic equations, which can be solved on a computer. The discretization phase consists of two steps: discretization of the computational domain and the discretization of the equations. The third phase is the solution of the algebraic equations. OpenFOAM uses finite volume method as a discretization method.

Finite volume discretization [54] of the computational domain provides a numerical description of the domain, including the positions of points in which the solution is obtained and the description of the boundary. The space is divided into a finite number of discrete regions, called control volumes. For time-dependent simulations, the time interval is divided into a finite number of time-steps. The discretization of the equations gives an appropriate transformation of terms of governing equations into algebraic expressions, which is the main goal of the discretization process. An iterative solution approached is used because the underlying physical phenomena are complex and non-linear. One of the most popular solution procedures is PISO algorithm, which stands for pressure implicit with splitting of operators [52]. Post-processing is visualization the solution domain in such a way to provide an insight of the flow motions.

Modularity and flexibility are the main advantages of OpenFOAM code. One of the modules is the linear algebra kernels module, which can be seen in almost all the PDE solver codes. This work selects two solvers from the OpenFOAM solvers set. One of them is icoFoam, which solves the incompressible laminar Navier-Stokes equations using the pressure-implicit splitting operator (PISO) algorithm iteratively. The other one is laplacianFoam, which solves the Laplace equation for unsteady, isotropic diffusion. Both solvers use internally linear solvers, which are supplied by the OpenFOAM main library. The linear solver, which is used in both icoFoam and laplacianFoam, is conjugate gradient (CG). However, the linear solver is selected at run-time. The following subsections provide essential information and structure analysis of icoFoam, laplacianFoam, and conjugate gradient solvers.

### 2.2.1 icoFoam

The incompressible laminar Navier-Stokes equations can be solved by icoFoam, which is an application supplied by the OpenFOAM standard solvers set. It applies the PISO algorithm in time stepping loop. The governing equations are the incompressible continuity 2.1 and momentum equations 2.2 as follows [52, 55]:

$$\nabla \cdot u = 0 \tag{2.1}$$

$$\frac{\partial u}{\partial t} + \nabla \cdot (uu) - \nabla \cdot (\nu \nabla u) = -\nabla p \tag{2.2}$$

Here $u$ is the vector velocity, $\nu$ kinematic viscosity, and $p$ is the pressure [52]. The coupling between the density and pressure is removed in incompressible flow, because the density changes are negligible. There is no prognostic pressure equation, but the continuity equation imposes a scalar constraint on the momentum equation; this combination can be used to derive a diagnostic equation for the pressure [52]. The Pressure Implicit with Splitting of Operators (PISO) algorithm solves the incompressible flow iteratively, which is described in the following section.

#### PISO Algorithm

The algorithm is essentially a predict-and-correct procedure for calculation of pressure on the collocated grid, where the flow quantities are defined at a single node in the control volume [52]. This type of grid arrangement can lead to well-known difficulties when coupling the pressure and velocities. A remedy for this problem is to use staggered grid for the velocity field [56]. Thus, to create a staggered grid, OpenFOAM uses a new variable that saves the interpolated velocity values at the faces.

The control-volume discretized Navier-Stokes equations for incompressible flow are as follows [55]:

$$a_c u_c = -\sum_n a_n u_n + \frac{u^*}{\Delta t} - \sum_f S(p)_f \tag{2.3}$$

$$\sum_f S.\left[\left(\frac{1}{a_c}\right)_f \nabla p_f\right] = \sum_f S.\left(\frac{H(u)}{a_c}\right)_f \tag{2.4}$$

Equation 2.3 is abstracted by introducing the *H(u)* term as in Equation 2.5, which contains the transport part including the matrix coefficients for all neighbors multiplied by the associated velocities [1], and the source part including the transient term [55]. We can rewrite Equation 2.3 as in Equation 2.6.

$$H(u) = -\sum_n a_n u_n + \frac{u^*}{\Delta t} \tag{2.5}$$

$$a_c u_c = H(u) - \sum_f S(p)_f \tag{2.6}$$

Here $a_n$ in Equation 2.3 is the matrix coefficient corresponding to the neighbors $n$ and $a_c$ is the central coefficient. The subscript $f$ implies the value of the variable (in Equation 2.3, it is $p$) in the middle of the face and $S$ is the outward-pointing face area vector. Equation 2.7 computes the face flux, *F*. The fluxes should satisfy the continuity equation.

$$F = S.u_f = S.\left[\left(\frac{H(u)}{a_c}\right)_f - \left(\frac{1}{a_c}\right)_f (\nabla p)_f\right] \tag{2.7}$$

The algorithm consists of three stages [55]. The first stage is the momentum predictor, which solves the momentum equation by using an initial or previous pressure field. This solution of the momentum equation gives the velocity field that is not divergence free but approximately satisfies the momentum equation. The second stage is the pressure solution that formulates the pressure equation and assembles the *H(u)* term. Third is the explicit velocity correction stage. Equation 2.7 gives set of conservative fluxes consistent with the new pressure field. Therefore, the velocity field is corrected as a consequence of the pressure distribution. The velocity correction is performed in an explicit manner [55] using Equation 2.8, which consists of two parts: $\frac{H(u)}{a_c}$ and $\frac{1}{a_n}\nabla p$ [55]. The first part is the transported influence of corrections of neighboring velocities. The second one is a correction due to the change in the pressures gradient. The main velocity error comes from the error in the pressure term, however, it is necessary to correct the *H(u)* term, formulate the new pressure equation, and repeat the procedure again [52, 54].

$$u_c = \frac{H(u)}{a_c} - \frac{1}{a_n}\nabla p \tag{2.8}$$

To sum up, the solution for incompressible flow problem obtained by icoFoam can be summarized as follows [52]:

1. Initialize all field values (from initial conditions or previous time step).
2. Start the calculation of the new time-step.
3. Formulate and solve the momentum predictor equation with the available face fluxes.
4. Invoke the PISO procedures until the tolerance for pressure-velocity system is reached, which is usually $1.0e-06$. At this stage, pressure and velocity fields for the current time-step are obtained, as well as the new set of conservative fluxes.
5. Using the new obtained values, solve all other discretized equations in the system.
6. Repeat from 2 until the last time step.

### 2.2.2 laplacianFoam

The solver is used to find the solution of the Laplacian equation. The equation contains one variable, a passive scalar, for instance, a temperature, $T$. The matrix is formed and computed [1]:

$$\frac{\partial T}{\partial t} - \nabla^2(D_T \cdot T) = 0 \tag{2.9}$$

Here $D_T$ is a constant [52]. laplacianFoam is one of the simplest OpenFOAM solvers. It is useful as starting point to understand and benchmark the linear solver. The linear solver that is used to solve the temperature in laplacianFoam as well as the pressure field in icoFoam is the Conjugate Gradient.

### 2.2.3 Linear Solvers

The result of the discretization process is a system of algebraic equations [52]. The matrices derived from the partial differential equations are large and sparse. The iterative methods avoid factorization, and instead perform matrix-vector operations and minimize the residual over the resulting vector space. These procedures are often referred to as Krylov subspace methods.

#### Conjugate Gradient Solver

One of the best-known Krylov sub-space methods is Conjugate Gradient [57], which is used to solve a system of linear equations given by the following form:

$$Ax = b \tag{2.10}$$

Here $x$ is an unknown vector of size number of cells in the computational domain, $b$ is a corresponding length vector, and $A$ is a known square, symmetric, positive-definite matrix. A pseudo-code of the common conjugate gradient algorithm as listed in Algorithm 1.

**Algorithm 1** Conjugate Gradient Method [57]

```
1: **Data:** Square, symmetric, and positive-definite matrix $A$, initial guess for vector
   $x$, constant vector $b$, and tolerance $\tau$
2: **Result:** Solution of linear system $x$
3: $p = r = b - Ax$
4: $rs_k = dotProduct(r, r)$
5: $norm_0 = initialNorm = computeNorm(r)$
6: $k = 0$
7: **while** $norm_k \geq \tau$ $and$ $k \leq max$ $number$ $of$ $iterations$ **do**
8:     $q = SparseMatrixVectorMultiply(A, p)$
9:     $\alpha = rs_k/dotProduct(p, q)$
10:    $x = x + \alpha * p$
11:    $r = r - \alpha * q$
12:    $norm_{k-1} = norm_k$
13:    $norm_k = computeNorm(r)$
14:    $rs_{k+1} = dotProduct(r, r)$
15:    $\beta = rs_{k+1}/rs_k$
16:    $p = r + \beta * p$
17:    $rs_k = rs_{k+1}$
18: **end while**
```

The behavior of the conjugate gradient method has been extensively studied. It is clear that the matrix vector multiplication is the most computationally intensive step in the algorithm [57]. The parallel-run of the conjugate gradient on a distributed memory architecture using MPI parallelizes the computations and introduces communications as well. Two types of communications are necessary to support the algorithm. The first type is point-to-point communication during the matrix-vector multiplication in order to include the influence of the parallel interfaces with neighbor processors. The second one is a collective communication to reduce the sum of scalars as part of forming inner products. It occurs three times while calculating global variable and updating the residual norm.

Several have proposed alternatives to conjugate gradients that reduce the global communications in order to improve the scalability of the parallel run. A recent study

[3] defines a pipeline conjugate gradient, which reduces the global communication into only one reduction per loop body instead of three. The pipeline conjugate gradient pseudo-code is as listed in Algorithm 2.

**Algorithm 2** Pipeline Conjugate Gradient Method [3]

```
1: **Data:** Square, symmetric, and positive-definite matrix $A$, initial guess for vector $x$, constant vector $b$, and tolerance $\tau$
2: **Result:** Solution of linear system $x$
3: $r_0 = b - Ax$
4: $w_0 = Ar_0$
5: $k = 0$
6: while $\|r\|_2 \geq \tau$ and $k \leq$ max number of iterations do
7:     $\lambda_k = dotProduct(r_k, r_k)$
8:     $\delta = dotProduct(w_k, r_k)$
9:     $q = SparceMatrixVectorMultiply(A, w_k)$
10:    if (k >0) then
11:        $\beta = \lambda_k / \lambda_{k-1}$
12:        $\alpha = \lambda_k / (\delta - (\beta * \lambda_k)/\alpha)$
13:    else
14:        $\beta = 0$
15:        $\alpha = \lambda_k / \delta$
16:    end if
17:    $z = q + \beta * z$
18:    $s = w + \beta * s$
19:    $p = r + \beta * p$
20:    $x = x + \alpha * p$
21:    $r = r - \alpha * s$
22:    $w = w - \alpha * z$
23: end while
```

The pipeline conjugate gradient as shown in algorithm 2 introduces more computational steps, unnecessary in the original, while reducing the global communication into one at computing the dot product step [3]. Given the trends of computer architecture, this is a very useful option, as we demonstrate.

## 2.3 Summary

Most CFD packages, as in our case OpenFOAM, are aimed at homogenous systems and mostly carried out on modern high performance platforms. However, the underlying architecture of these platforms have taken significant turns. Several hybrid and heterogeneity-aware implementations have been done [26]. Recently, there have been few attempts [45, 43] to implement hybrid MPI+OpenMP CFD code. The OP2 [51] project provides a framework to implement CFD application using unstructured meshes on different computational hardware including multi-core and multi-GPU system. However, OP2 is not applicable for multi-block codes as in the case of OpenFOAM. One of the major works in the heterogenous computing area is the functional performance model, which distributes the workload across the heterogenous devices in accordance to the processing elements speed [58].

# Chapter 3

# Parallel Computing in OpenFOAM

A common approach for parallelizing numerical algorithms is domain decomposition. The mesh and its associated fields are divided into subdomains and allocated to separate processors. OpenFOAM uses the domain decomposition to perform a process level parallelism using the runtime library MPI. It provides different utilities of domain decomposition such as METIS [46] and SCOTCH [47]. The communication between the subdomains is explicitly handled within the package. This chapter explains how the parallelism is handled in OpenFOAM and presents a performance analysis of the selected solvers. Section 3.1 describes its parallel model followed by a description of the domain decomposition methods available in OpenFOAM. Section 3.2 explores the Cufflink [4] library that extends some of the OpenFOAM linear solvers capabilities to be computed on GPU. Section 3.3 presents the performance analysis results of OpenFOAM selected solvers.

## 3.1 OpenFOAM in Parallel

The domain decomposition approach is a natural way to make good use of parallel computing in CFD [52, 88]. The main idea is to subdivide the domain into subdomains and assign each one to a processor. There is a one-to-one mapping between the processors and the subdomains, thus, it is a single program multiple data parallelism

(SPMD). In this case, each processor executes the same program on its own set of data. At some stage each processor needs data that exists in other subdomains, then communication between processors is performed to synchronize the data.

OpenFOAM applies domain decomposition to enable process-level parallelism [1]. Thus, the program can run in parallel on separate subdomains, with communication between processors using the MPI communications protocol. The first step to enable a parallel run in OpenFOAM-based application is to decompose the solution domain into subdomains. The number of subdomains should be equal to the number of running processors. The parallel run engages communication and synchronization between the processors on each subdomain. There are mainly two types of communication: communication with neighboring, and global communication with all the processors. As with most mesh based applications, the communication scheme between processors is based on the halo-layer approach with overlapping elements, which duplicates the cells next to a processor boundary (internal boundary). The halo-layer covers all internal boundaries and is explicitly updated through communications between processors.

In OpenFOAM, the parallelization strategy employs a zero-halo layer [55, 1], which considers the processor boundaries as internal edges and treated as boundary conditions. It introduces implicit updated boundaries, thus, the difference between parallel and serial run is the presence of processor boundary. The parallel communications are wrapped in a *Pstream* module, which isolates the communications details from the code of icoFoam or any other solver in OpenFoam. Thus, similar patterns can be seen in some local computation that requires an interface influence to be included. This common pattern can be summarized in the following stages: initialize the interfaces, perform local computations, and then update interfaces as well as include the interfaces influence into the computations. The communication wrapper *Pstream* contains processor rank and size, and performs both point-to-point communication and collective ones in generic forms. Also it contains global mesh information, and handles patched neighboring communication. One processor owns each cell in the computational subdomain. The faces can be categorized as: internal faces located within a subdomain, inter-processor boundary faces that make up the border of two subdomains, and external boundary faces, where physical boundary conditions are applied.

### 3.1.1 Domain Decomposition Methods

OpenFOAM supports four methods of domain decomposition, which decompose the data into non-overlapping subdomains [1, 2]. These methods are Simple, Hierarchical, METIS [46], and SCOTCH [47]:

- **Simple**: The simple geometric decomposition, as its name implies, splits the domain into pieces by coordinate in a tensor product fashion, e.g. $n$ cuts along the $x$ axis and $n$ cuts along $y$ axis. The user should specify the number of partitions per each coordinate in the space.

- **Hierarchical**: Hierarchical geometric decomposition is similar to the simple except the user specifies the order in which the directional split is done recursively, e.g. first in the $y$ direction then the $x$ direction.

- **METIS**: METIS is a versatile library for partitioning and ordering matrices and graphs. It is used by OpenFOAM to partition the computational domain. By default OpenFOAM enables multilevel k-way partitioning to minimize the communication volume, however, the user may specify either of the two main methods of

METIS, which are multilevel recursive bisections and multilevel k-way partitioning. Both methods try to minimize the communication volume between processors. Also, it is possible to specify weights for the different subdomains. The parallel version of METIS is called ParMETIS.

• **SCOTCH**: SCOTCH is a library for partitioning and ordering graph and mesh and attempts to minimize the number of internal boundaries. The domain decomposition can be modeled as graph partitioning problems on the adjacency graph of matrices. The main purpose is to separate the edges in a way to cause the minimal communication between internal boundaries. This library implements the multilevel FM-like algorithms, k-way graph partitioning and recursive bi-partitioning. The user can specify the strategy and the weights of the subdomains. Also, it offers a parallel version called PtSCOTCH.

## 3.2 Cufflink Library

Cufflink [4] stands for CUDA-For-FOAM-Link, it is an open source library that implements an interface between OpenFOAM and numerical methods based implementation on NVIDIA's Compute Unified Device Architecture (CUDA) [78]. In other words, Cufflink solves the linear system derived from OpenFOAM using CUSP [5] and Thrust [59] library calls. It implements two linear solvers, conjugate gradients and bi-conjugate gradients stabilized. Both solvers are implemented based on CUSP routines and Thrust library support. Cufflink supports Multi-GPUs via OpenFOAM process-level parallelism through domain decomposition. The parallel-run of Cufflink assumes homogenous processor capabilities that each processor has a GPU to accelerate the linear system. For example, if the computational domain has been decomposed into 4 subdomains, then, 4 GPGPU kernels will be assigned, one-to-one, to accelerate them.

The internal implementation of Cufflink is based on CUSP and Thrust libraries. CUSP is a C++ template library for sparse matrix and graph computations on CUDA architecture GPUs [5]. It offers a flexible API, and high-level interface for manipulating sparse matrices. It supports most common sparse matrix formats such as coordinate (COO), compressed sparse row (CSR), diagonal (DIA), ELLPACK (ELL) and Hybrid (HYB) [84, 85]. Each format has advantages and disadvantages, which depend on the matrix structure of the problem. CUSP implements most of the main sparse matrix operations such as add, subtract, copy convert, and multiply. The Thrust library is a parallel algorithm library that resembles the C++ Standard Template Library (STL) on CUDA architecture GPUs [59]. Cufflink extends the conjugate gradient solver and bi-conjugate gradient stabilized solver in OpenFOAM to be run on CUDA architectures, but does not support hybrid computation, MPI+GPGPU.

## 3.3 Performance Analysis and Results

In order to accurately assess the profiling of the selected solvers, we turned off the I/O feature and carried out all simulations to avoid any possible noise. The selected solvers consist of several time-steps, however, only the second time-step has been scanned in order to avoid any start up allocation noise at the first time-step. The selected platform (Tesla), which is used to benchmark and profile the applications, is a single node consisting of two sockets, each socket is connected to a GPU device. The socket has 6 dual cores Intel Xeon CPU X5670 @ 2.93GHz. Figure 3.1 shows the topology of the platform by invoking *lstopo* command that renders the topology of the machine [63]. Table 3.1 shows the GPU devices characteristics. However, the profiling is for CPU computation only because there is no hybrid application to profile.

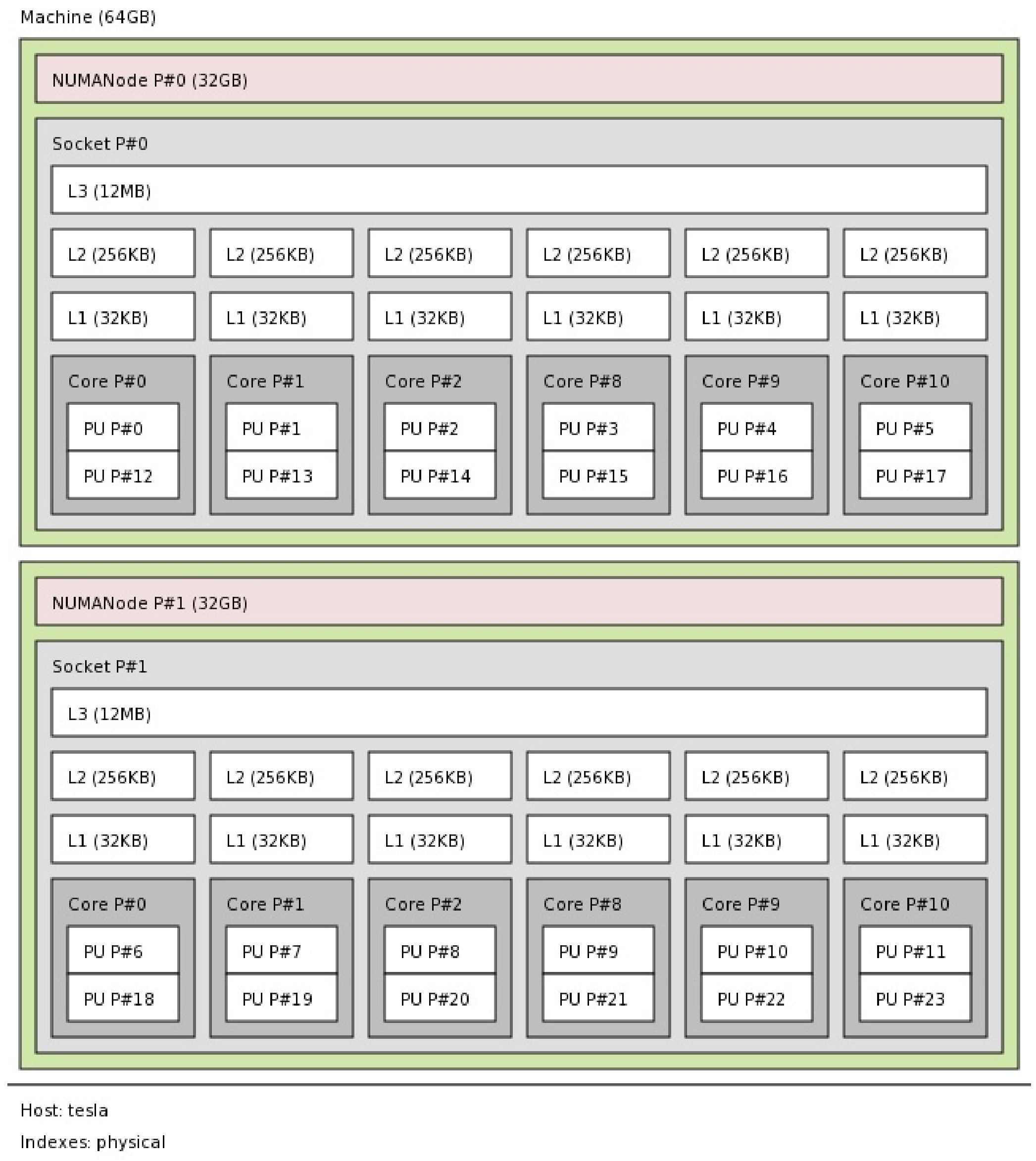


Figure 3.1: Topology of Tesla machine by invoking *lstopo*.

Table 3.1: Tesla Node GPU Devices

| |
|---|
| Device 0: “Tesla C2050” |
| CUDA Driver Version / Runtime Version 5.5 / 5.5 |
| CUDA Capability Major/Minor version number: 2.0 |
| Total amount of global memory: 2687 MBytes (2817982464 bytes) |
| (14) Multiprocessors x ( 32) CUDA Cores/MP: 448 CUDA Cores |
| GPU Clock rate: 1147 MHz (1.15 GHz) |
| Memory Clock rate: 1500 Mhz |
| Memory Bus Width: 384-bit |
| L2 Cache Size: 786432 bytes |
| Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048) |
| Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048 |
| Total amount of constant memory: 65536 bytes |
| Total amount of shared memory per block: 49152 bytes |
| Total number of registers available per block: 32768 |
| Warp size: 32 |
| Maximum number of threads per multiprocessor: 1536 |
| Maximum number of threads per block: 1024 |
| Maximum sizes of each dimension of a block: 1024 x 1024 x 64 |
| Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535 |
| Maximum memory pitch: 2147483647 bytes |
| Texture alignment: 512 bytes |
| Concurrent copy and kernel execution: Yes with 2 copy engine(s) |
| Run time limit on kernels: No |
| Integrated GPU sharing Host Memory: No |
| Support host page-locked memory mapping: Yes |
| Alignment requirement for Surfaces: Yes |
| Device has ECC support: Enabled |
| Device supports Unified Addressing (UVA): Yes |
| Device PCI Bus ID / PCI location ID: 2 / 0 |

Test cases have been chosen in order to analyze the performance of icoFoam and laplacianFoam. The test cases are:

• The lid-driven cavity flow test case [1] contains the solution of a laminar, isothermal and incompressible flow over a three-dimensional unstructured cubic geometry using the icoFoam solver. The top boundary of the cube is a wall that moves in the $x$

direction, whereas the rest are static walls. The size of the mesh is $100\times100\times100$ cells, Table 3.2 shows more information about the mesh obtained from calling checkMesh command supported by OpenFOAM.

•The other test case is the heat equation test case solved over a three-dimensional unstructured cubic geometry using the laplacianFoam. In addition, we studied a single iteration of the conjugate gradient solver using a matrix derived from the 3D heat equation.

In order to determine the optimal number of processors, several experiments were carried out, which have similar patterns, however, the eight subdomains were chosen, each four are bound to each socket. The SCOTCH method is used to decompose the solution domain into evenly distributed subdomains. The profiling tools, which are used to scan the applications and collect statistics, are Extrae and Paraver [62, 64], IPM [66] and TAU [65] for parallel-run and GNU profiler.

Table 3.2: The output of *checkMesh* on Mesh with a million cells.

| **Mesh Statistics** | **Value** |
|---|---|
| Points | 1030301 |
| Faces | 3030000 |
| Cells | 1000000 |
| Boundary Patches | 2 |

| **Patch Type** | **Faces** | **Points** |
|---|---|---|
| Moving Wall | 10000 | 10201 |
| Fixed Walls | 50000 | 50201 |

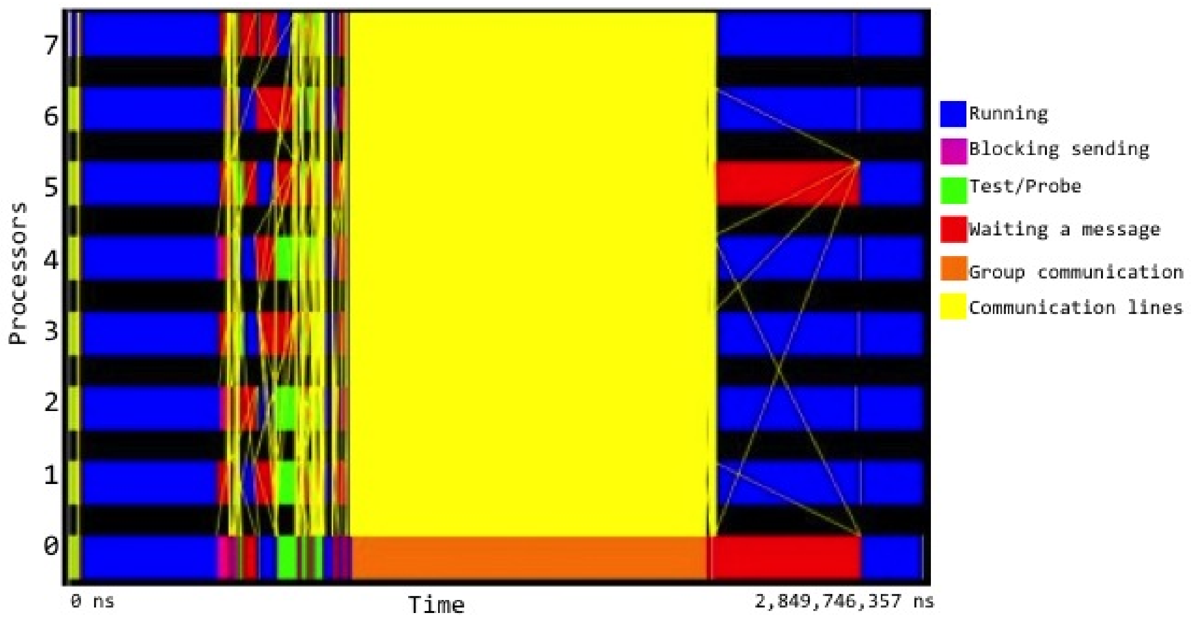


Figure 3.2: Trace of the second step laplacianFoam application executed using eight MPI processes on the Tesla Machine.

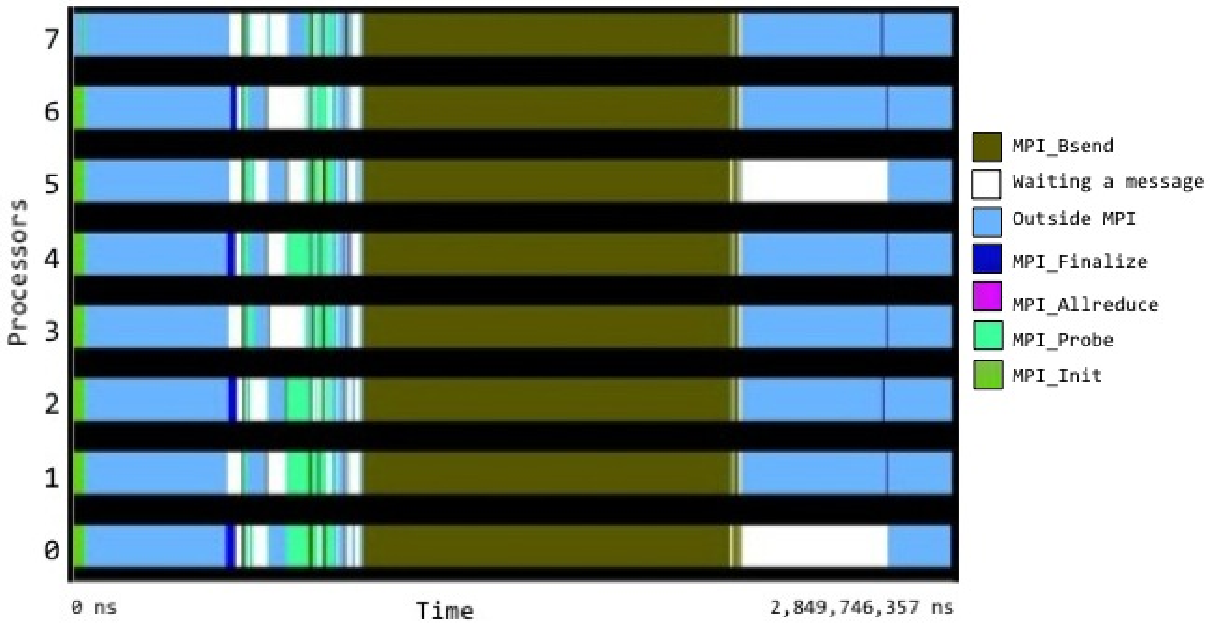


Figure 3.3: Timeline of MPI calls executed at each point in time by each process during the execution of the second step laplacianFoam application on the Tesla Machine

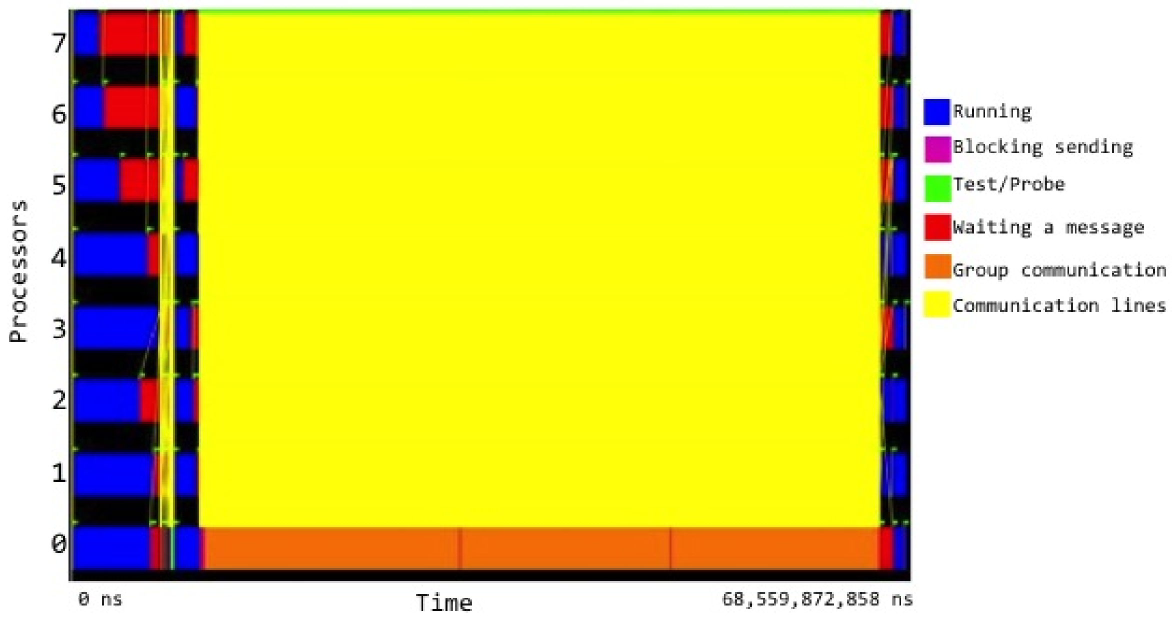


Figure 3.4: Trace of the second step icoFoam application on the Tesla Machine

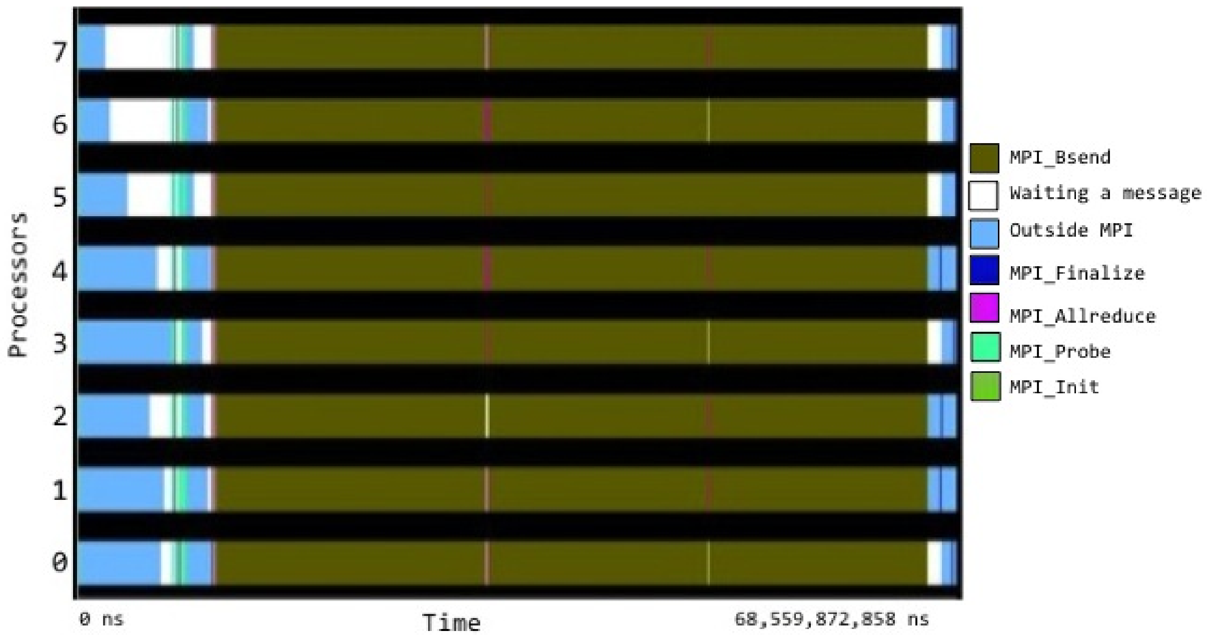


Figure 3.5: Timeline of MPI calls executed at each point in time by each process during the execution of the second step icoFoam application on the Tesla Machine

Figure 3.2 shows the trace of the second time-step of the laplacianFoam, which solves a 3D heat equation using the OpenFOAM conjugate gradient linear solver internally to solve the temperature field. The horizontal axis represents the total execution time of the application. The color encoding is as follows.

- The blue represents the computation on each process.
- The red indicates the processor is waiting, i.e., idle.
- The orange represents the global communication. In this case, it symbolizes MPI-Allreduce calls.
- The green and pink represent the point-to-point communication phase i.e. send and receive.
- The yellow lines represent communication between processors.

Figure 3.3 shows the timeline of which each MPI call is being executed at each point in time by each process. The horizontal axis represents time, from the start time of the application at the left of the figure to the end time at the right. For every processor, the colors represent the MPI call or light blue when doing computation outside of MPI. The white color indicates idle time where the processor is waiting. This execution consists of several hundreds iterations of the conjugate gradient solver. Similarly, Figure 3.4 shows the trace of the second time-step of the icoFoam application, and, Figure 3.5 presents the timeline of which the MPI call is being executed at each point in time by each process. The horizontal axis represents the total execution time of the second time-step execution of icoFoam. It solves the 3D cavity test case using PISO algorithm iteratively. The dominance of the global communication is strong in icoFoam. The MPI-Allreduce occupies more than 75% of MPI time as shown in Figure 3.6 that has been obtained obtained by IPM [66]. However, as a laplacianFoam trace file, this run consists of several hundreds of iterations of the conjugate gradient loop. Thus, we scanned a one iteration of the conjugate gradient solver using a matrix derived from the 3D heat equation as shown in Figure 3.7.

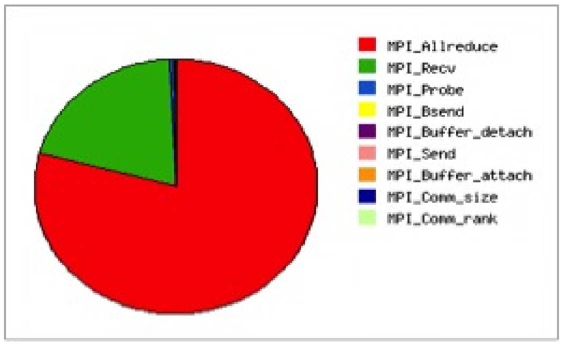


Figure 3.6: Profile result obtained by IPM that shows the percentage of MPI-time consumed by each MPI call

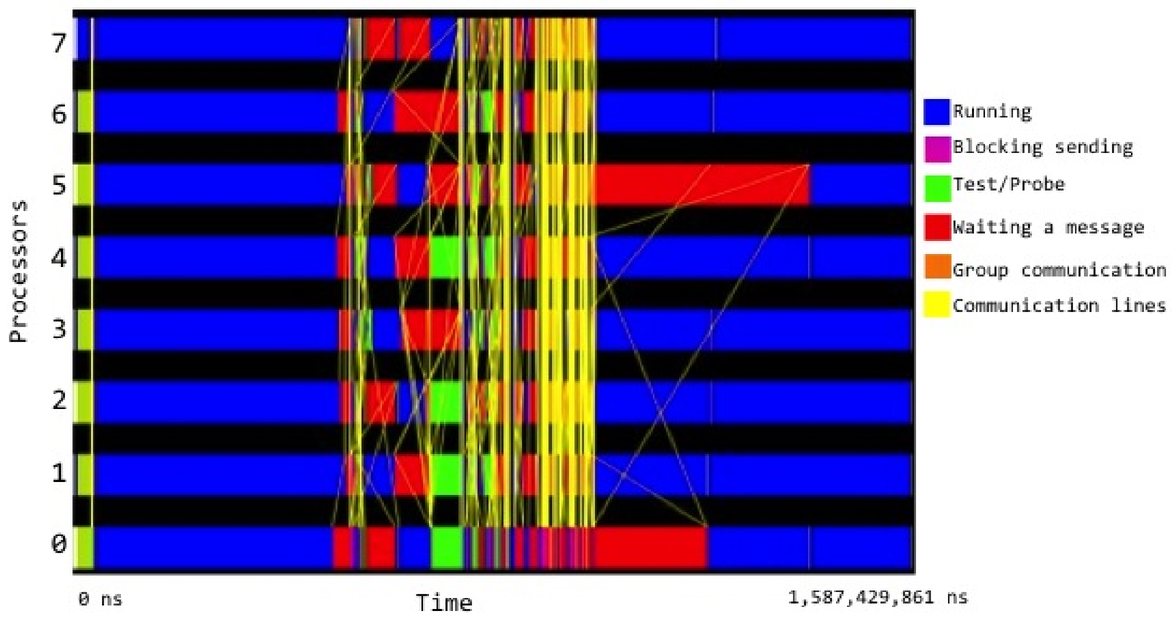


Figure 3.7: Trace of one iteration of the conjugate gradient linear solver

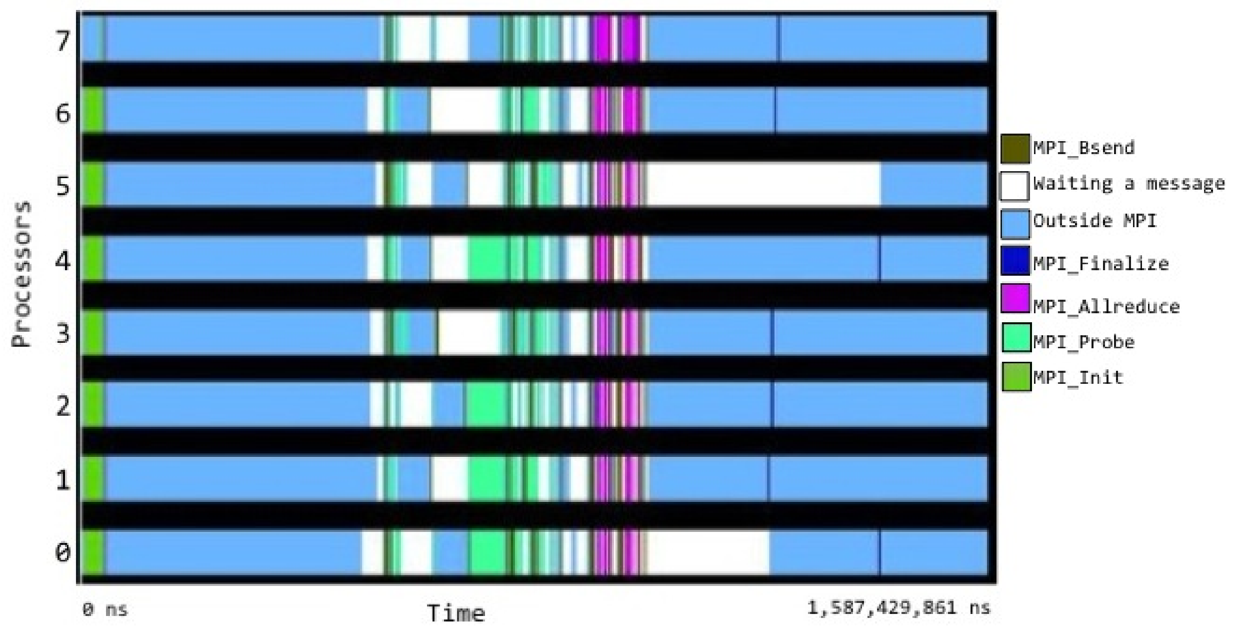


Figure 3.8: Timeline of MPI calls executed at each point in time by each process while executing an iteration of CG

Figure 3.7 shows the profiling trace of the conjugate gradient (CG) solver implemented in OpenFOAM main library version 1.6-ext. The horizontal axis represents the total execution time of one iteration of CG. From the figure, we can observe the effect of the collective communication MPI-Allreduce and the point-to-point communication, mainly the receive call. The percentage of MPI calls is 31% of the total execution time. Figure 3.8 presents the timeline of which MPI call is being executed at each point in time by each process while executing an iteration of CG. The colors represent the MPI calls or light blue when doing computation outside of MPI. From the profiling result, the CG main computation phase is the matrix-vector multiplication, which can be seen at the left of the Figure 3.7.

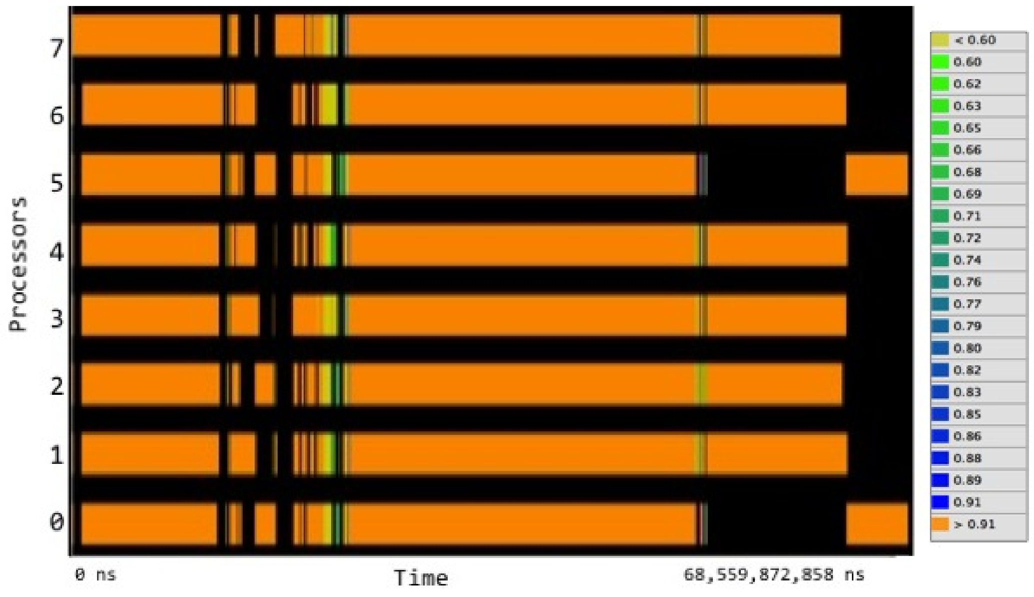


Figure 3.9: Timeline with the instructions per cycle (IPC) achieved by each interval of useful computation during the execution of laplacianFoam

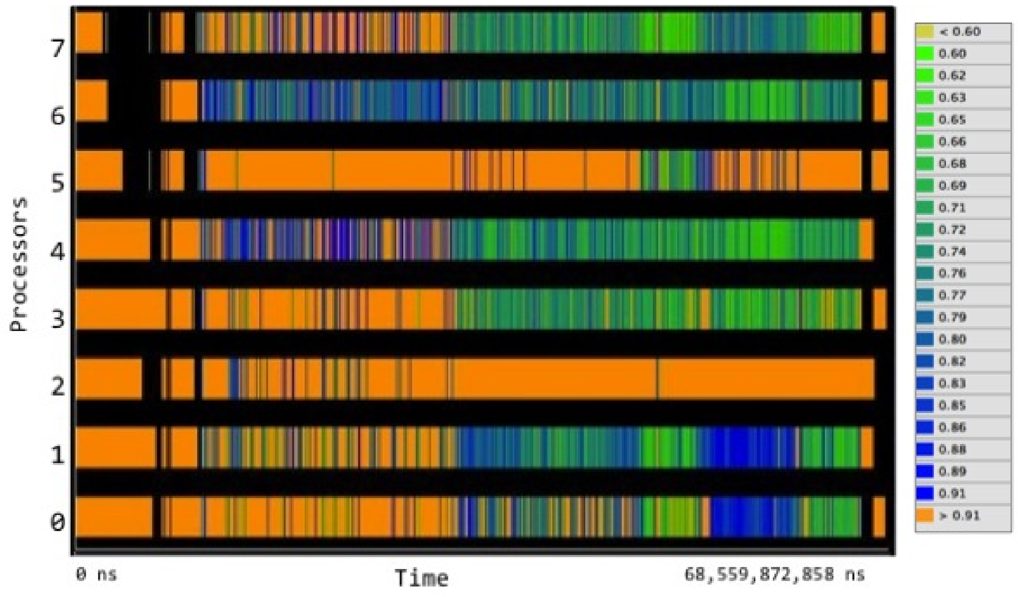


Figure 3.10: Timeline with the instructions per cycle (IPC) achieved by in each interval of useful computation during the execution of icoFoam

Figure 3.9 shows a timeline with the instructions per cycle (IPC) achieved by each interval of useful computation during the execution of laplacianFoam. Also, Figure 3.10 shows a timeline with the IPC achieved in each interval of useful computation during the execution of icoFoam. The color encoding of IPC function of time for each process is a gradient between light green representing a low IPC value and dark blue representing a high IPC value. This view shows in black in the regions where processes are in MPI in order to focus on the actual useful computation parts, however, this configuration of Paraver does not take into account the collective communications. Some part of the computation phases show low IPC value due to high cache misses rate. Taking a closer look to OpenFOAM structures, it is not optimized for cache blocking, thus it causes high cache miss rates. The same observation can be seen in icoFoam profiling result as well, however, a high IPC value is indicated towards the end of the graph, this is associated with the conjugate gradient phase. From the profiling results, one of the possible optimization option is enhancing the structures and the data storage to be more cache friendly. This step requires a magnificent amount of editing that time and topic restrains do not allow for here, however, it can be noted as future work.

## 3.4 Summary

The OpenFOAM parallel run is based on MPI level parallelization. The domain decomposition plays an important role in the parallel run. If the domain is not decomposed properly according to the computing devices and minimizing the boundaries between the processors, the parallel run will suffer from an unbalanced workload execution and communications overhead. According to the performance analysis, the collective communication is the dominant part in MPI calls. Minimizing the global communication and hiding the necessary calls would offer better scalability and speed-up. The most time consuming part as expected is the sparse linear algebraic kernels.

# Chapter 4

# Hybrid Heterogeneous Solver

The main motivation of this work is to increase the performance of the selected solvers and maintain an effective use of the allocated resources. The benchmarking shown in Figure 4.1 suggests that the CPUs performance is good enough to be included in the computation side by side with the GPUs. The available parallelization in OpenFOAM is either: MPI on multi-core, or GPU on multi-GPU. Figure 4.2 presents the available ways to accelerate the linear solver kernel. Although both OpenFOAM CG and Cufflink CG implement the same algorithm, the code structure can not be combined to produce the hybrid code due to deadlock.

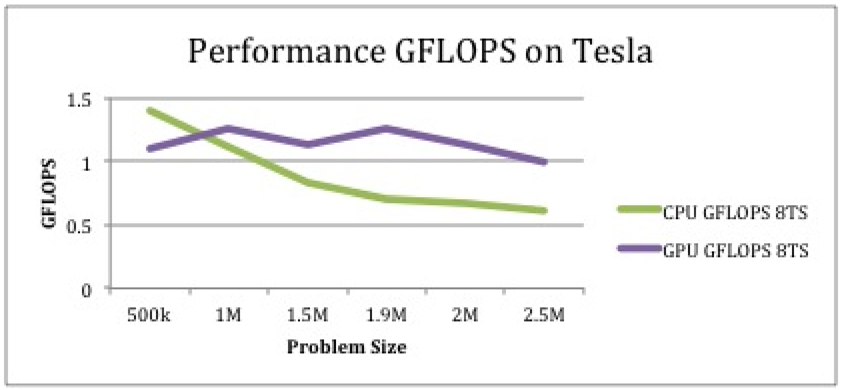


Figure 4.1: Performance of Cufflink CG and OpenFOAM CG. icoFoam has been used to benchmark both solvers, while solving the pressure field of 3D cavity test case.

As shown in Figure 4.1, icoFoam has been executed to benchmark the OpenFOAM

CG linear solver and Cufflink CG. The test case is a lid-driven cavity flow over a 3D cubic mesh. We measured the total execution time of the entire run. We observed that Cufflink CG converges slower than OpenFOAM CG, although the precision in both linear solvers is double precision. A closer look at the Cufflink CG code, there are many memory copy from host to device and vice versa.

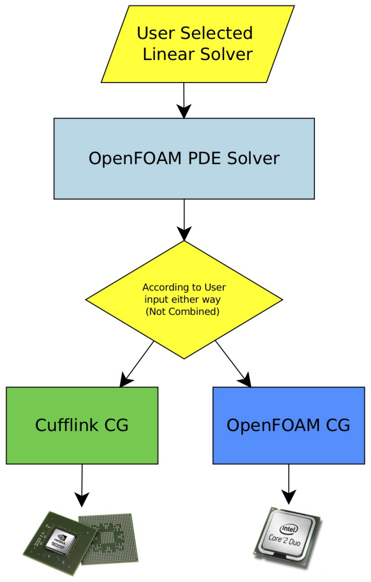

User Selected Linear Solver
OpenFOAM PDE Solver
According to User input either way (Not Combined)
Cufflink CG
OpenFOAM CG

Figure 4.2: Available parallel code of CG integrated in OpenFOAM. Cufflink provides only multi-GPU code, and OpenFOAM provides only multi-core code. Both solvers can not be combined because of the different communication scheme.

The first solution algorithm proposed in this thesis is the hybrid conjugate gradient, which solves the system of linear equations derived from the partial differential equations using the original conjugate gradient method on heterogeneous hybrid platforms efficiently. Besides developing the hybrid conjugate gradient solver, which is an engineering problem, there are two main challenges that have been also addressed in this work: how to distribute the data across highly heterogeneous devices, which is addressed in Section 4.3, and how to minimize the synchronization points. The latter challenge is addressed by implementing the pipeline conjugate gradient algorithm, which is an alternative algorithm of the conjugate gradient method [3]. Accordingly, the second proposed solver is the pipeline conjugate gradient, which has been implemented in three versions: GPGPU only, MPI only, and hybrid MPI+GPGPU.

## 4.1 Hybrid Conjugate Gradient

The main idea is to distribute the computation across heterogeneous devices some of them being GPUs and run the computation in parallel. All the processors employ a process-level parallelism using MPI and each MPI-process, which is connected to a GPU, accelerates the computation using CUDA. Those processors, which are not connected to a GPU run the computation locally.

From the development point of view, the hybrid conjugate gradient has been implemented and designed as a set of functions, which mimics the original design in OpenFOAM and the same linking interfaces as in Cufflink. However, the communication scheme is hybrid, and, has been redesigned from scratch. Figure 4.3 sketches the design and implementation of the hybrid Conjugate Gradient solver.

We used CUSP and Thrust library for implementing CUDA kernels. Figure 4.3 shows the computational flow on the CPU and on the GPU. It started by decomposing the computational domain into $p$ subdomains using OpenFOAM *decomposePar* function, which invokes the selectable domain decomposition method to decompose the mesh according to a given number of subdomains. This step is considered as a pre-processing step and requires input from the user such as the number of subdomains, and the domain decomposition method. As mentioned earlier, OpenFOAM provides four types of domain decomposition methods (DDM): Simple, Hierarchical, METIS, and SCOTCH. The last two support static load balancing partitioning algorithms, which minimize the communications between neighbors.

The application repeatedly invokes the linear solver, which is, in our case, the hybrid CG. The number of CPUs and GPUs involved in the execution may vary during the parallel execution, e.g., 2 GPUs + 6 CPUs. The communication scheme of one iteration requires three global communications and one point-to-point communication between neighboring subdomains. The matrix is derived from the discretized equation, sparse, and of the dimension of the number of finite volumes. In case of decomposed computational domain, the local matrix row dimension is number of cells within the subdomain. The hybrid CG solver supports common sparse matrix storage formats, which are: coordinate (COO), compressed sparse row (CSR) and hybrid (HYB), and, it is selectable by the user.

As shown in Figure 4.2, the available implementations of CG in OpenFOAM are: the OpenFOAM CG, which is parallelized using MPI, and Cufflink CG, which is parallelized using GPGPU. Although both codes implement the original CG algorithm, their CG functions can not be combined directly due to different communication schemes, which could cause deadlock at some stages. For instance, collecting information about the parallel interfaces causes deadlock, because, there is a delay in the Cufflink code due to copying the data from host to device. Therefore, it is impossible to combine the two codes to solve the problem on two subdomains correctly. We implemented the conjugate gradient method to be able to run in multi-core/multi-GPU platform.

Let us consider a computational domain that has been decomposed into two subdomains. Our platform is a dual-core socket connected to one GPU device. A configuration file, which is given, selects processor with rank 0 to be accelerated. At the start of the linear solver phase, OpenFOAM interface *lduMatrix* [1] contains the coefficient matrix, the vector $b$, and mesh related information. The matrix storage consists of three arrays: lower diagonal, diagonal, and upper diagonal matrix. This object will be passed to the hybrid CG solver and initialize it. According to the given configuration file, each MPI process makes a decision to call the CUDA kernel or the CPU kernel. In this example processor $p_0$ will call the CUDA kernel, whereas, processor $p_1$ will call the CPU kernel. During the initialization phase of CUDA kernel, the required data is copied from the host to device. At the beginning, both kernels collect information about the processor boundaries (parallel interfaces) such as neighbors ranks and volume of data required to be send/receive to/from neighboring using point-to-point communication between adjacent subdomains. The procedure flow continues as shown in Figure 4.3.

The user may provide a configuration file, which contains the information about the processors, their location with respect to each other, and which socket is connected to which GPU. This file will be used by the hybrid solver to bind the computations to the processor according to its location and accelerate the processor that is nearby the GPU. This step is important to allocate the data to the processor in a hardware-aware topology that adequately map communicating processes onto the processors of the platforms.

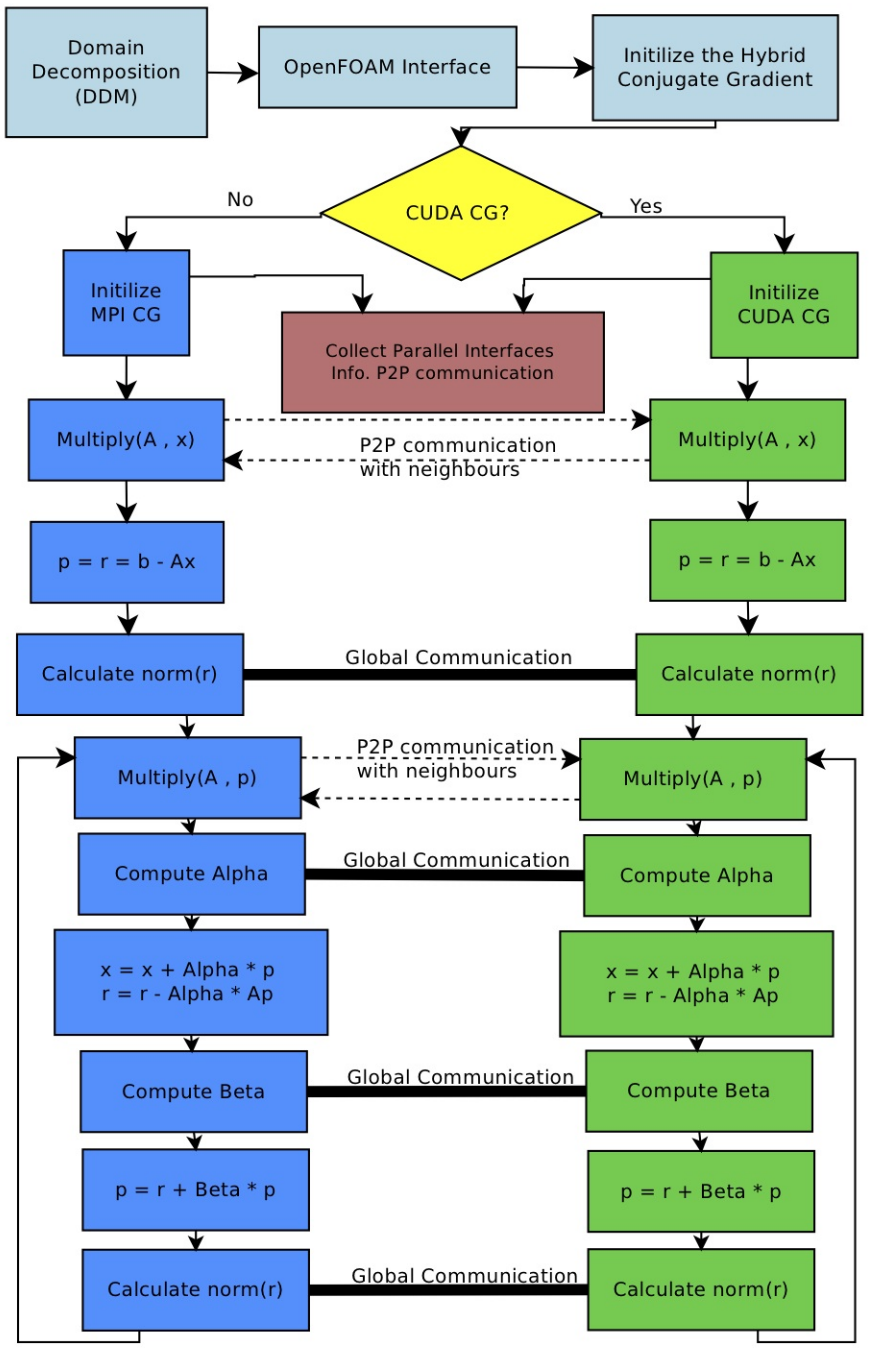


Figure 4.3: Design and implementation of the hybrid Conjugate Gradient solver, showing the steps executed on CPUs on the left (in blue), and, the steps executed on GPUs on the right (in green), and the coordination presents through black arrows.

## 4.2 Hybrid Pipeline Conjugate Gradient

We implemented the pipeline conjugate gradient algorithm, which is motivated by the formerly mentioned work of Ghysels and VanRoose [3]. It can be considered as a communication start up reducing algorithmic improvement of the original method. The pipeline conjugate gradient minimizes the global communication to only one collective communication per loop body instead of three. It offers opportunity of better scalability, however, at the price of extra computations, which are relatively cheaper. Figure 4.4 sketches the design and implementation of the hybrid Pipeline Conjugate Gradient solver. It shows the computational flow on the CPU and on the GPU.

The pipeline conjugate gradient has been designed and implemented in three versions: GPGPU/CUDA, MPI, and hybrid MPI+GPGPU/CUDA. The algorithm, which is listed in Algorithm 2, requires one reduction operation per iteration, which is during the computation of $\lambda$ and $\delta$. The MPI only implementation stores the matrix as OpenFOAM *lduMatrix* format, which stores the matrix in three arrays: lower, diagonal, and upper matrix storage. As the matrix is symmetric only the lower is stored. We implemented the linear algebra operations in our library with cache blocking mechanism, that is why we did not use OpenFOAM operations.

The GPGPU pipeline conjugate gradient has been implemented in CUDA. This version of pipeline CG consists of code to be run on a host CPU, and CUDA kernel that consists of lines of code to be run on GPU device. The CUDA initialization block in Figure 4.4 begins communicating with other subdomains in order to collect information about the parallel interfaces between processor boundaries. After that, the host prepares the required data to be delivered to the device. We used Thrust storages and the copy function to send the data from host to device and conversely from device to host. The procedure flow continues as listed in Algorithm 2. The inner implementation of the linear algebra operations is based on CUSP library. It supports common sparse matrix storage formats, which are: coordinate (COO), compressed sparse row (CSR) and hybrid (HYB), and, it is selectable by the user.

The hybrid pipeline CG combines the MPI pipeline CG and CUDA pipeline CG kernels. There are two phases in Algorithm 2: outside loop phase, and the loop phase. The loop phase is more important than the other one, because it iterates several times. The first communication inside the loop is a single MPI_Allreduce that compute the sums of the global *lambda* and *gamma*. It is followed by matrix-vector multiplication, which requires talking with neighboring subdomains to update the processor boundary. The point-to-point communication can be between two hosts, two CPUs kernel, or hybrid. These communications require data movements between host and device that can be overlapped with local computations of the parallel interfaces. This optimization has been implemented in order to reduce the cost of communication and data movements. The procedure flow continues as listed in Figure 4.4. The inner implementation is combination of the other two implementations functions i.e. MPI pipeline CG and CUDA pipeline CG.

All the three pipeline conjugate gradient solvers and the hybrid conjugate gradient solver have been compiled and integrated into OpenFOAM as a dynamic library.

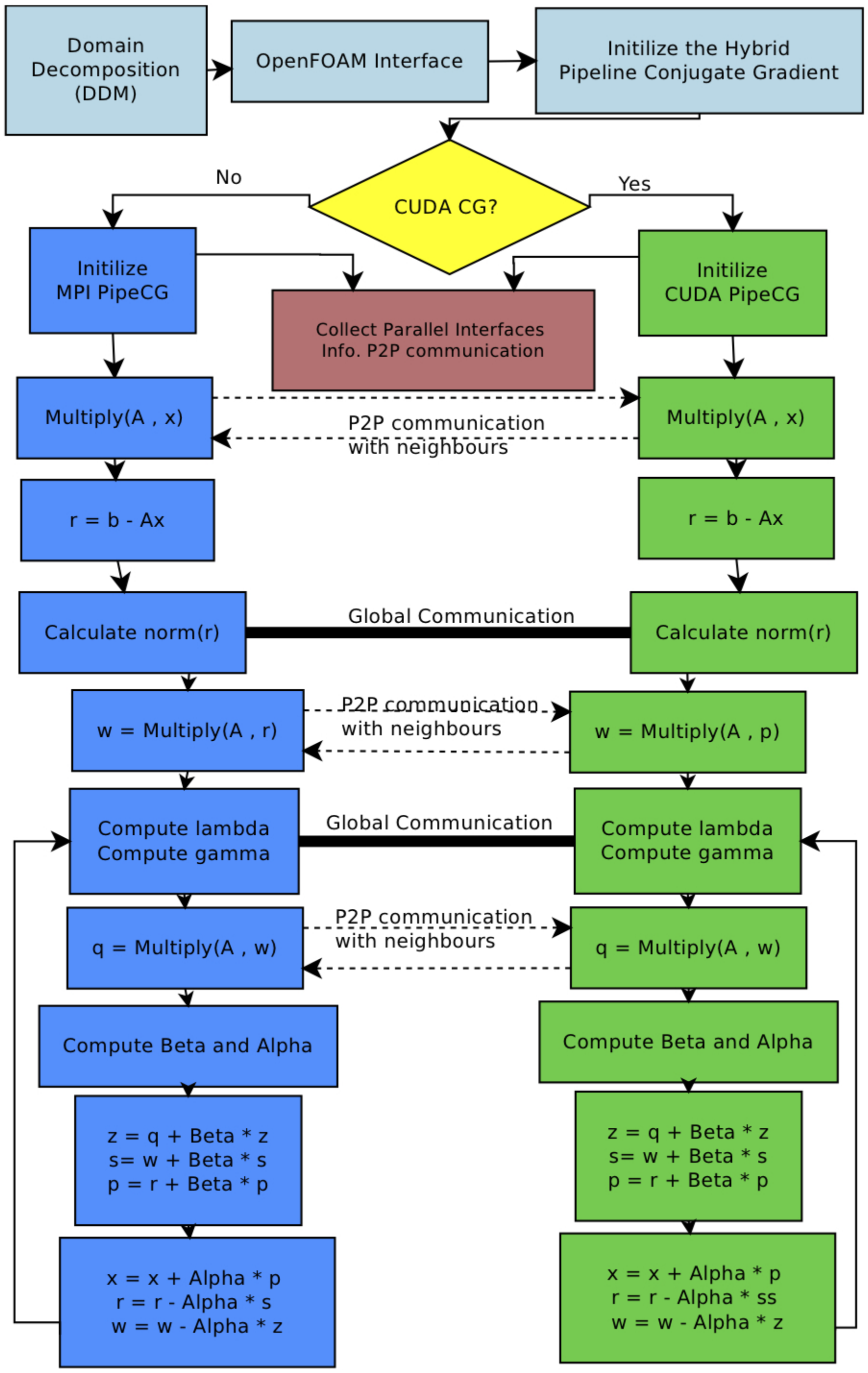

Domain Decomposition (DDM)
OpenFOAM Interface
Initilize the Hybrid Pipeline Conjugate Gradient
CUDA CG?
No
Yes
Initilize MPI PipeCG
Initilize CUDA PipeCG
Collect Parallel Interfaces Info. P2P communication
Multiply(A , x)
P2P communication with neighbours
Multiply(A , x)
r = b - Ax
r = b - Ax
Calculate norm(r)
Global Communication
Calculate norm(r)
w = Multiply(A , r)
P2P communication with neighbours
w = Multiply(A , p)
Compute lambda Compute gamma
Global Communication
Compute lambda Compute gamma
q = Multiply(A , w)
P2P communication with neighbours
q = Multiply(A , w)
Compute Beta and Alpha
Compute Beta and Alpha
z = q + Beta * z s= w + Beta * s p = r + Beta * p
z = q + Beta * z s= w + Beta * s p = r + Beta * p
x = x + Alpha * p r = r - Alpha * s w = w - Alpha * z
x = x + Alpha * p r = r - Alpha * ss w = w - Alpha * z

Figure 4.4: Design and implementation of the Hybrid Pipeline Conjugate Gradient solver; interpreted as in Figure 4.3.

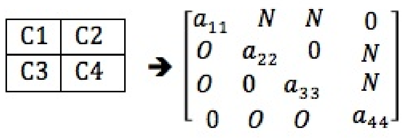
## 4.3 Heterogeneous Decomposition

In the case of the hybrid solver, an even distribution of the data is out of the question, because it will cause workload imbalance between the accelerated computation using GPGPU and the sequential computations. The main idea behind the heterogeneous decomposition is to adequately partition and assign the subdomains to the processors in proportion to their performance. Also, the technique should take into account minimizing the communications, so as to minimize inter-processor communications and network congestion.

As described in previous sections, OpenFOAM provides static load balancing during the decomposition process through some third-party library, that is, METIS, ParMETIS [46], SCOTCH or PtSCOTCH [47] prior to running the program. METIS and SCOTCH both allow the user to provide a constant number per processor that represents the percentage of data to be computed by this processor. However, neither library provides any methods or algorithms to compute this percentage for balanced workload. It is up to the user to supply these indicators.

In a nutshell, the heterogeneous decomposition combines the performance model and the METIS/SCOTCH library. It estimates the relative speed of each compute device in the target platform by running the computational kernel on evenly distributed subdomains, and measures the execution time per processor. Then, it computes the speed of each processor, and builds the performance model of the application. The output of this step is vector of numbers representing the computational volume per-processor accordance to its speed. Hence, the load of the processor will be balanced if the number of computations performed by each processor is accordance to its speed on execution the kernel. After that, the percentage will be provided as an input to METIS/SCOTCH to re-decompose the data.

Let us assume the hybrid conjugate gradient solver is the computational kernel to be optimized using the heterogenous decomposition. The input is a square sparse matrix $A_i$, which contains $n_i \times n_i$ elements, where $n_i$ is number of cells allocated to processor $p_i$. Each CG iteration requires one matrix-vector multiplication, two dot products, and several scalar multiplications and vector additions. Number of non-zeros in $A_i$ is $n_i + 2F_i$, where $F_i$ is number of faces in the subdomain $i$ that represents number of the off-diagonal elements. As in [86, 13], the computational complexity of CG kernel is *O*($n_i + 2F_i$). In other words, the number of operations to process one non-zero element of this matrix can be considered as a constant, therefore, the amount of computation to process matrix $A_i$ is proportional to number of non-zeros that is, $n_i + 2F_i$. As mentioned before, the matrix is given as an input from OpenFOAM matrix interface, with the number of cells and number of faces stored in the matrix object. A simple example of OpenFOAM matrix is shown in Figure 4.5 that illustrates a coefficient matrix, which has been formed from a 2D cubic mesh consisting of 4 hexahedra cells. The cell with the lower label is called the owner of the face that the face area vector is constructed in such a way that it points outwards the owner cell [55].



$$\rightarrow \begin{bmatrix} a_{11} & N & N & 0 \\ 0 & a_{22} & 0 & N \\ 0 & 0 & a_{33} & N \\ 0 & 0 & 0 & a_{44} \end{bmatrix}$$

Figure 4.5: Example of simple 2D mesh and its corresponding coefficient matrix

The speed $s_i$ of processor $p_i$ during computing the CG kernel is $n_i + 2F_i$ divided by the execution time $T$, which can be formulated in Equation 4.1. Then, the relative speed $r_i$ is computed according to Equation 4.2. After that, the new assigned number of cells to be allocated at processor $p_i$ is computed using Equation 4.3. This vector of $p$ numbers will be passed as argument to METIS/SCOTCH API to re-decompose the computational domain correspondingly.

$$s_i(n_i) = \frac{n_i + 2F_i}{T(n_i)} \tag{4.1}$$

$$r_i(n_i) = \frac{s_i(n_i)}{\sum_p s_i(n_i)} \tag{4.2}$$

$$n_{i,new} = N * r_i(n_i) \tag{4.3}$$

The following diagram shows the heterogeneous decomposition.

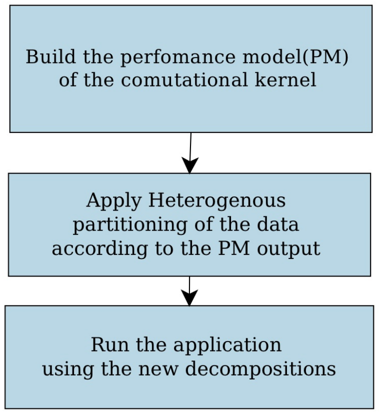

Build the perfomance model(PM)
of the comutational kernel
Apply Heterogenous
partitioning of the data
according to the PM output
Run the application
using the new decompositions

Figure 4.6: Design of the heterogeneous decomposition

METIS and SCOTCH both are designed to find minimum internal boundaries between subdomains, hence less communications between processors. Combining their design with the performance model that reflects the ability of each processor to compute with respect to other processors will obtain a balanced workload and minimal communications partitioning. However, the CFD applications may consists of different computational kernels, which requires more auto-tuning using the heterogenous decomposition that time constraints do not allow for here. This step is noted for future work, and the current implementation is a stepping stone towards balancing computational kernels across highly heterogenous devices. In this work, we try to optimize the most computational intensive phase in the application, and show how this optimization can accelerate the whole applications. However, more enhancements are possible to increase the effectiveness of the decomposition. These enhancements are noted as future work.

One more challenge to be addressed is to allocate the data to the processor in a hardware-aware topology that adequately map communicating processes onto the processors of the platforms. The user may provide a configuration file, which contains the information about the processors and their location with respect to each other and which socket is connected to which GPU. This file will be used by the hybrid solver to bind the computation to the processor according to its location and accelerate the processor that is nearby the GPU.

## 4.4 Summary

This chapter presents the design and implementation of the hybrid solvers and the heterogeneous decomposition. The motivation behind the hybrid solver is to include the CPUs in the computation with GPUs that is, MPI+GPGPU. The hybrid approach has been designed and implemented for the conjugate gradient method and the pipeline conjugate gradient method. A better decomposition is essential in the case of the hybrid solver. Therefore, we proposed the heterogeneous decomposition method. This method is implemented on top of METIS and SCOCH that are shipped with OpenFOAM as third-libraries. The main idea is to balance the computation by decomposing the data using METIS/ SCOTCH with performance model that reflects each processor capability. The heterogenous decomposition and the hybrid solvers have been integrated and linked into OpenFOAM.

# Chapter 5

# Evaluation

This chapter presents the main experimental results obtained in our work. The previous chapter discusses the design of the proposed code optimizations. There are four optimizations to be evaluated, which are: the hybrid conjugate gradient solver, the hybrid pipeline conjugate gradient solver, the heterogeneous decomposition, and the pipeline conjugate gradient GPU and MPI only solvers.

## 5.1 Experimental Platforms

Our library is implemented in C and C++ and plugged into OpenFOAM accordingly. Our version of OpenFOAM is OpenFOAM-ext1.6. The experimental platforms are Tesla machine and HCL cluster. The platforms are described as follows.

- **Tesla**: It is a single node consisting of two sockets; each socket is connected to GPU device. One socket has 6 dual cores Intel Xeon CPU X5670 @ 2.93GHz. Figure 3.1 shows the topology of the platform by invoking *lstopo* command that renders the topology of the machine [63]. Table 3.1 shows the GPU device characteristics.

- **HCL cluster**: A comprehensive description from the HCL cluster [61] wiki is quoted here. “The HCL cluster is heterogeneous in computing hardware and network ability. Nodes are from Dell, IBM, and HP, with Celeron, Pentium 4, Xeon, and AMD processors ranging in speeds from 1.8 to 3.6Ghz. Accordingly architectures and parameters such as Front Side Bus, Cache, and Main Memory all vary. Operating System used is Debian with Linux kernel 2.6.32." [61] Diagram 5.1 shows a schematic of the cluster.

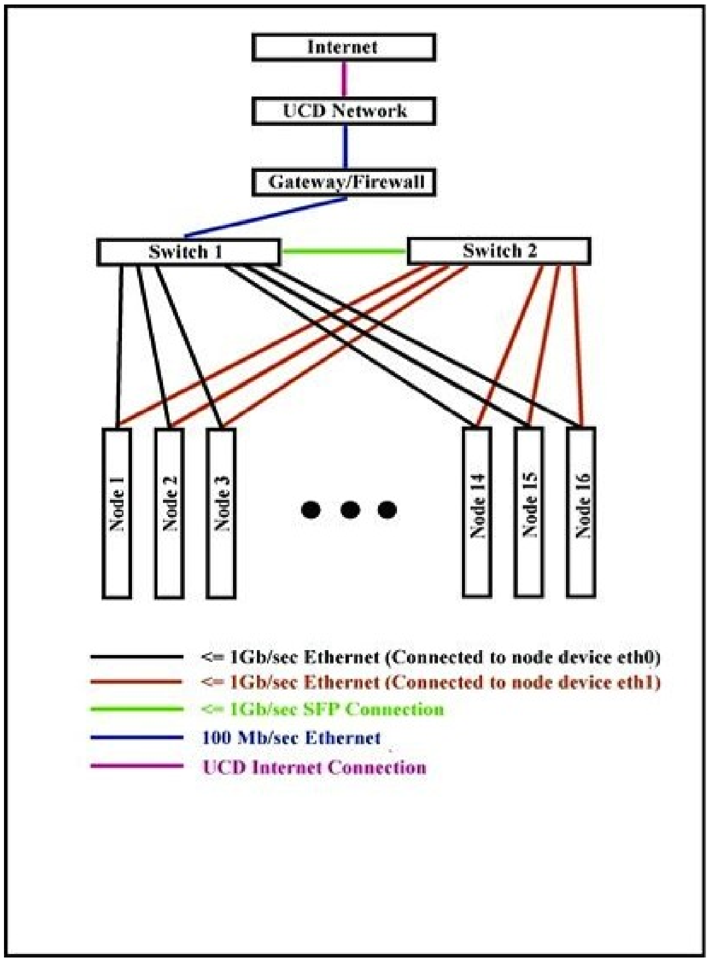


Figure 5.1: Diagram shows the underlying architecture topology of HCL cluster

## 5.2 Test Cases

- **3D lid-driven cavity flow case**: The lid-driven cavity flow [2] test case contains the solution of a laminar, isothermal and incompressible flow within a three-dimensional unstructured cubic geometry using the icoFoam solver. The top boundary of the cube is moving wall that moves in the $x$ direction, whereas the rest are static walls.

- **3D heat equation case** [2]: The heat equation test case is solved over a three-dimensional unstructured cubic geometry using the laplacianFoam.

The number of cells in the mesh varies from 500,000 to 2,000,000 cells, generated using the OpenFOAM mesh utility *blockMesh* [1]. In this utility, we choose hexahedra as the cell type.

## 5.3 Results and Discussions

We benchmarked both OpenFOAM conjugate gradient, which is parallelized using MPI, and Cufflink conjugate gradient, which is implemented using CUDA. The benchmark is the 3D lid-driven cavity flow case on cubic mesh consisting of 1,000,000 cells. The case is solved using icoFoam running for 8 time-steps on Tesla. Figure 4.1 shows the result of parallel execution of the icoFoam solver for several problem sizes. Number of processors used is bound to number of available GPUs on Tesla machine, which are two GPUs. The precision used in the computation of the 3D cavity test case is double-precision. The conjugate gradient method called to solve the pressure field. According to Figure 4.1 the performance gap between the CPU and the GPU solvers is not significant. This means that CPUs are good to be included in the computation with GPUs in a hybrid approach.

### 5.3.1 Pipeline Conjugate Gradient Solvers

We implemented three variations of the pipeline conjugate gradient: hybrid Pipeline CG, CUDA Pipeline CG on GPU, and MPI Pipeline CG (CPU only). Figure 5.2

shows the performance of CUDA Pipeline CG and Cufflink CG on Tesla machine using 2 GPUs, and 2 hosts. The benchmark is solving system of linear equations that has been derived from 3D heat equation test case for several problem sizes. Using two GPUs with 2 hosts is a realistic environment to use on Tesla machine for both Cufflink CG and CUDA Pipeline CG, because, both solvers are for cluster of multi-GPU. Moreover, Figure 5.3 shows the speed-up of both CUDA Pipeline CG and Cufflink CG over the sequential execution of CG on Tesla machine for several problem sizes. The CUDA Pipeline CG consistently outperforms the Cufflink CG. The main point behind this speed-up is that Pipeline CG offers less communications, therefore, less data movements.

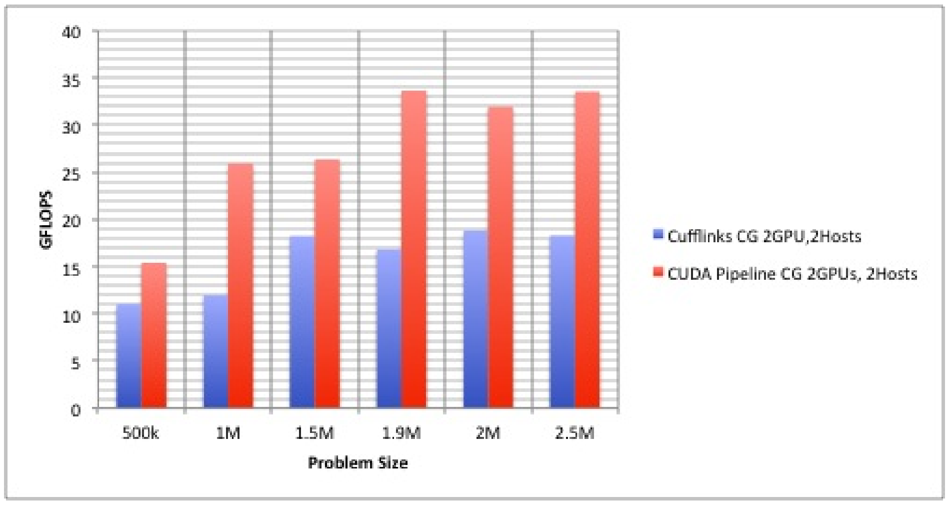


Figure 5.2: Performance comparison of the Cufflink CG and CUDA Pipeline CG (2GPUs and 2Hosts) on Tesla Machine.

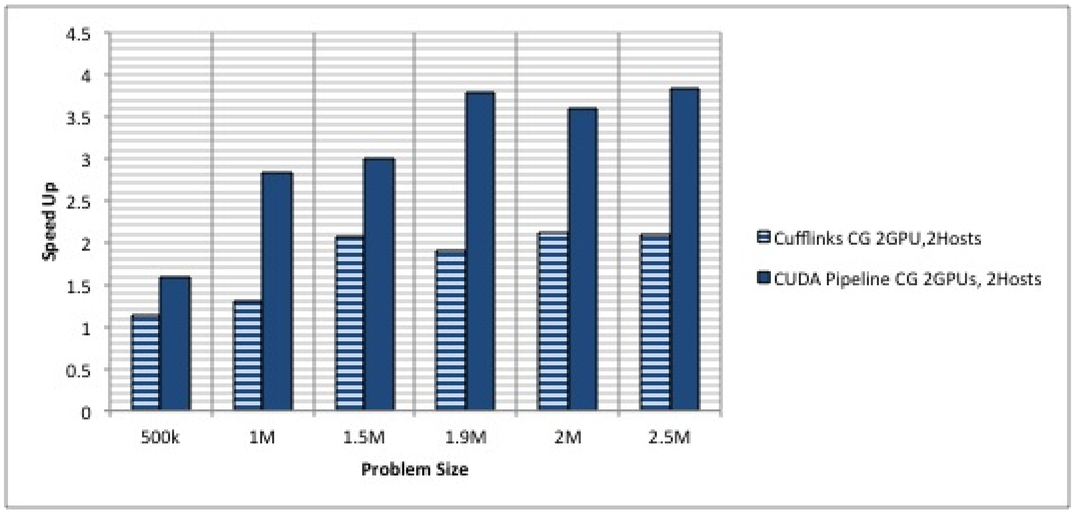


Figure 5.3: Speed-up comparison of the Cufflink CG and CUDA Pipeline CG (2GPUs and 2Hosts) relative to sequential CG on Tesla Machine.

Figure 5.4 shows the speed-up of the parallel MPI Conjugate Gradient provided by OpenFOAM and MPI Pipeline CG over the OpenFOAM sequential CG. This test case has been parallelized using 2 processors on Tesla machine. On the other hand, Figure 5.5 shows the performance comparison between OpenFOAM MPI CG and MPI Pipeline CG parallelized using 24 processors.

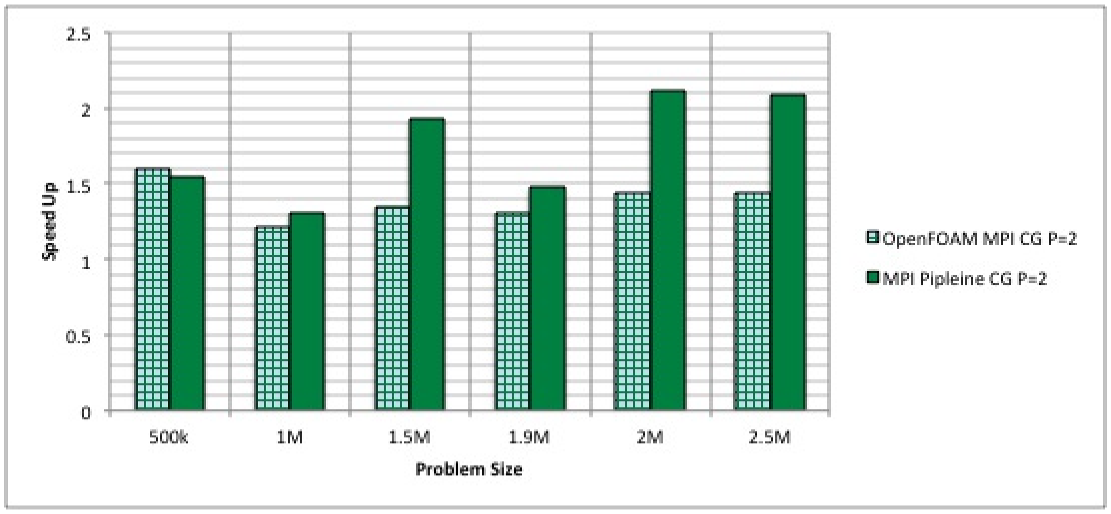


Figure 5.4: Speed-up comparison of the OpenFOAM Parallel CG and Parallel Pipeling CG (MPI only) using 2 processors on Tesla Machine, relative to sequential CG.

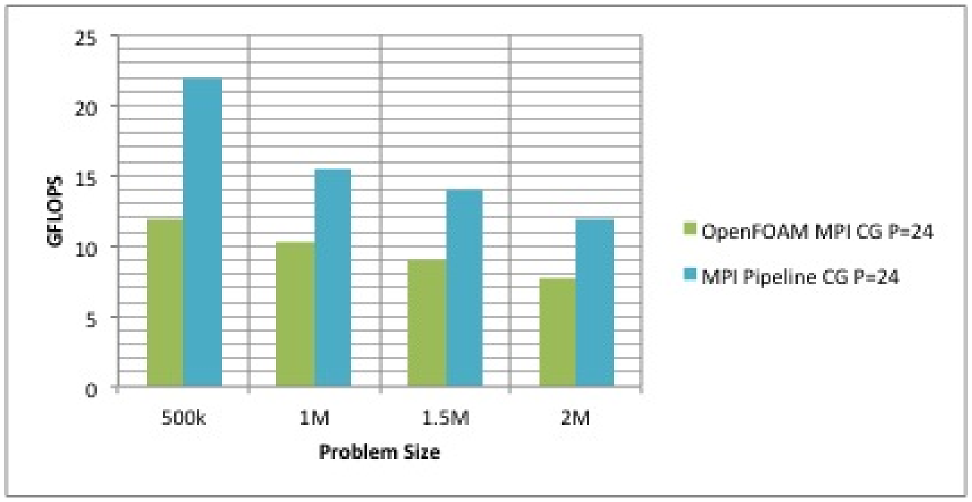


Figure 5.5: Performance comparison of the OpenFOAM Parallel CG and Parallel PipeCG (MPI only) using 24 processors on Tesla Machine

## 5.3.2 Heterogeneous Decomposition Results

We tested the heterogeneous decomposition on HCL cluster using OpenFOAM CG linear-solver called from icoFoam 20 time-steps (MPI only). The HCL cluster provides a heterogeneous platform that consists of processors with different performance. The objective of this test is to measure the effectiveness of the heterogeneous decomposition.

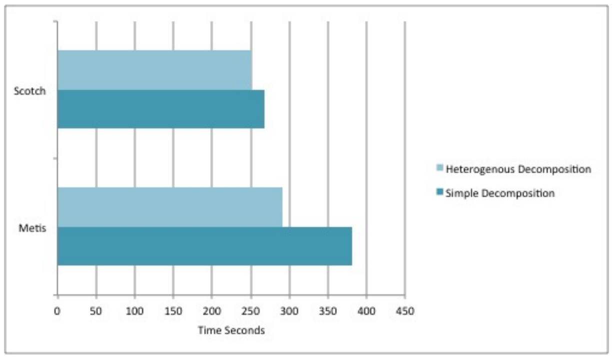


Figure 5.6: Total execution time comparison of the heterogeneous decomposition and the original decomposition for running parallel icoFoam 20 time-steps that calls OpenFOAM CG linear-solver on HCL cluster.

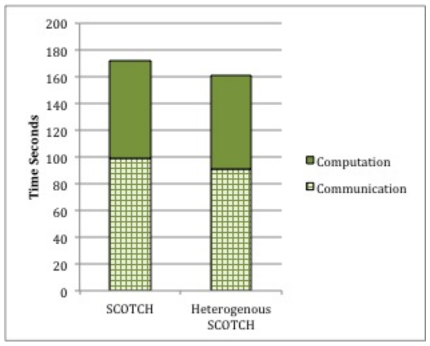


Figure 5.7: Total execution time comparison of the SCOTCH heterogeneous decomposition and the original decomposition for running parallel icoFoam 20 time-steps that calls OpenFOAM CG linear-solver on HCL cluster.

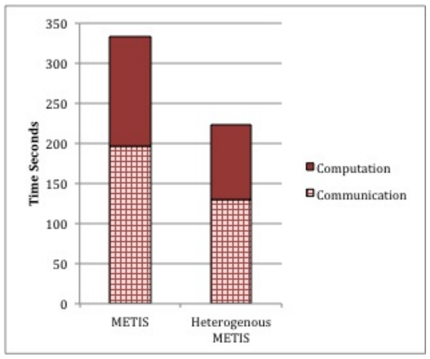


Figure 5.8: Total execution time comparison of the METIS heterogeneous decomposition and the original decomposition for running parallel icoFoam 20 time-steps that calls OpenFOAM CG linear-solver on HCL cluster.

The HCL cluster multiple levels of heterogeneity are: processors with different speed and different memory. The experiments involved 12 heterogeneous processors with the fastest processor being 25% faster than the slowest one. The heterogeneous decomposition is compared against the decomposition obtained by providing equal weights to the partitioner for all the subdomain, as shown in Figure 5.6. Figure 5.7 shows the comparison between the heterogenous SCOTCH decomposition against the SCOTCH one. Figure 5.8 shows around 25% improvement in heterogenous METIS against the non-modified one. Moreover, the heterogeneous decomposition is compared against equal decomposition on Tesla machine. This test uses the hybrid CG, which solves a system of linear equations derived from 3D heat equation for several problem sizes. The computational domain has been decomposed into two subdomains using SCOTCH decomposition method. One subdomain has been computed on a GPU, but the other one on a CPU. Figure 5.9 shows that the heterogenous SCOTCH decomposition provides consistent improvement of the total execution time around 25%.

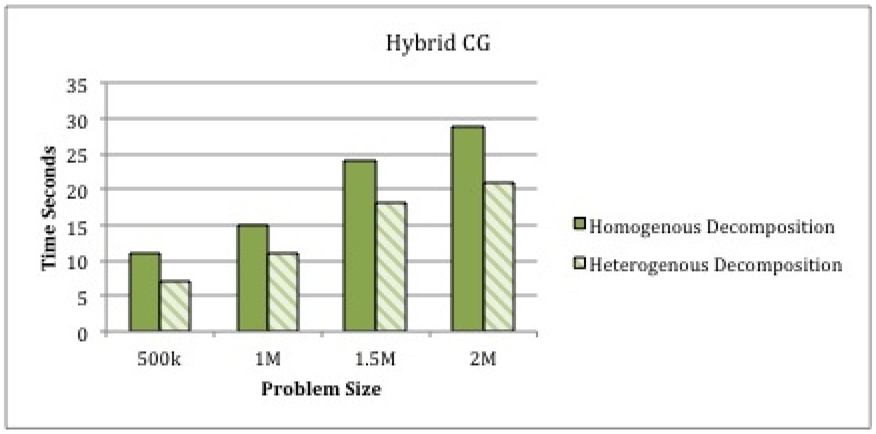


Figure 5.9: Total execution time comparison of the SCOTCH heterogeneous decomposition and the original decomposition for running parallel Hybrid CG linear-solver on Tesla Machine.

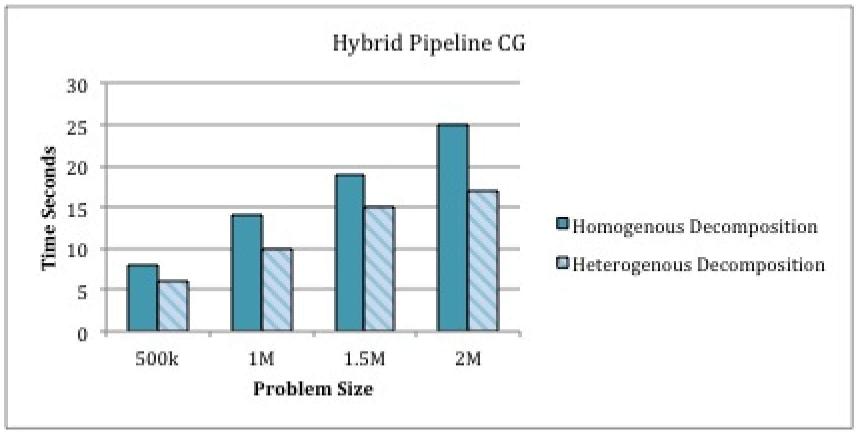


Figure 5.10: Total execution time comparison of the SCOTCH heterogeneous decomposition and the original decomposition for running parallel Hybrid PipeCG linear-solver on Tesla Machine.

### 5.3.3 Hybrid Solvers Results

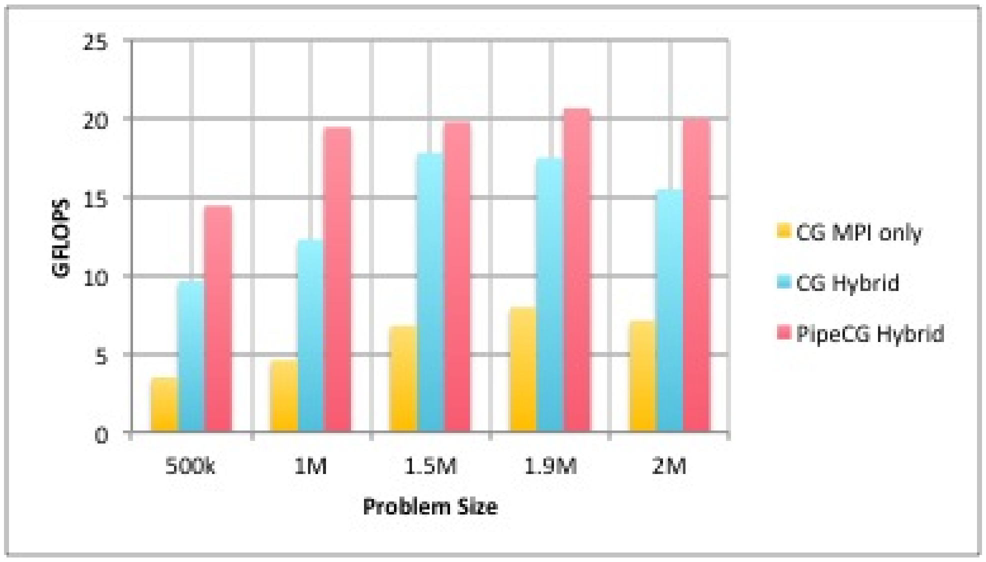


Figure 5.11: Performance comparison of OpenFOAM parallel CG, hybrid CG and hybrid pipeline CG on different problem size running on Tesla machine

Figure 5.11 compares hybrid CG and hybrid pipeline CG against OpenFOAM MPI CG. The test case that has been used is solving the system of linear equations derived form 3D heat equation for several problem sizes. The computational domain has been decomposed into 24 subdomains. The hybrid solvers accelerate the computations using 22 cores and 2-GPUs, 2-hosts. The average speed-up of hybrid CG against the OpenFOAM parallel CG is 1.7x, and hybrid pipeline CG provides 2x speed-up, as shown in Figure 5.11.

By optimizing only the CG linear solver, we manage to accelerate the whole execution of the application. We examine the speed-up and performance gain from optimizing the conjugate gradient solver on the end-to-end computation of the selected solvers: icoFoam and laplacianFoam. Figure 5.12 shows the speed-up of the laplacianFoam using different linear solvers over the sequential execution of laplacianFoam. The linear solvers are MPI Conjugate Gradient supported by OpenFOAM, CUDA Conjugate Gradient supported by Cufflink, MPI Pipeline Conjugate Gradient, and CUDA Pipeline Conjugate Gradient. From the figure, the average speed-up provided by OpenFOAM MPI CG using 2 processors is around 1.4x. Cufflink CG shows 1.28x average speed-up. The MPI Pipeline CG using 2 processors shows 1.68x average speed-up. The fastest one was the CUDA Pipeline CG, which shows 2.02x average speed-up. Furthermore, Figure 5.13 shows the speed-up of the total execution time of laplacianFoam against Cufflink GPU CG execution time. The two solvers are: hybrid CG and hybrid pipeline CG. The computational domain has been decomposed several times for different number of subdomain. Therefore number of processors, which is noted in the figure as P, varies from 2 to 8. From the figure, 2 subdomains shows the best configuration of the machine.

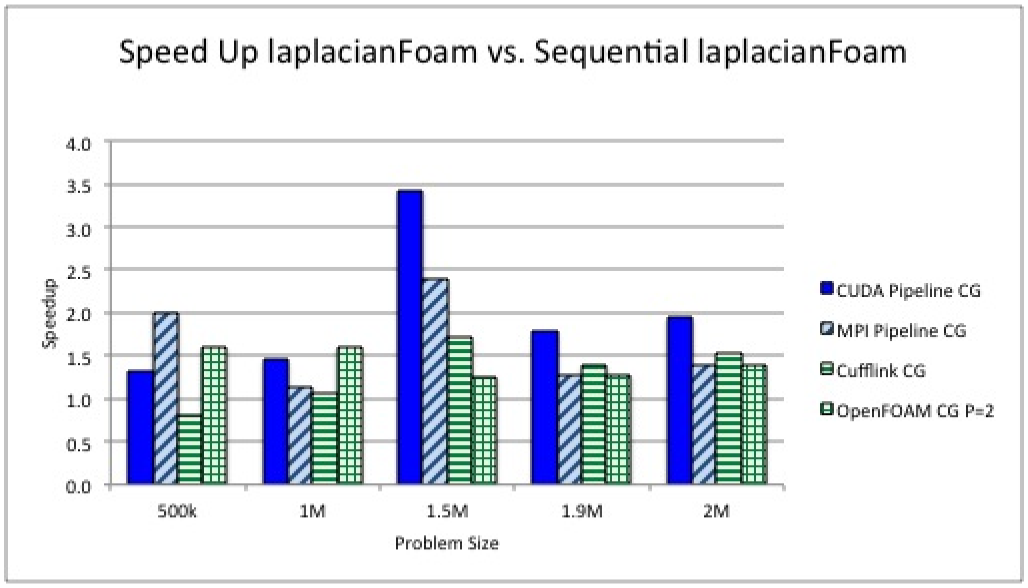


Figure 5.12: Speed-up of the execution time of different parallel run over the sequential execution of laplacianFoam

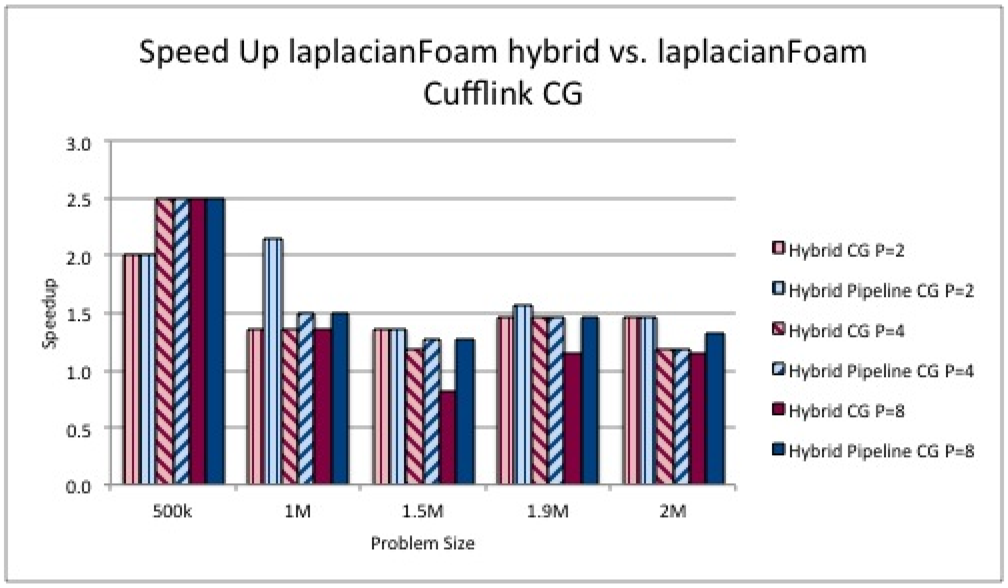


Figure 5.13: Speed-up of the execution time of different parallel run over the laplacianFoam calling Cufflink CG

Figure 5.14 shows the speed-up of the icoFoam using different linear solvers over the sequential execution of icoFoam. The linear solvers are CUDA Conjugate Gradient supported by Cufflink, MPI Pipeline Conjugate Gradient, and CUDA Pipeline Conjugate Gradient. The test case that has been used is the 3D cavity case for several mesh sizes. The computational domain has been decomposed into two subdomains. Figure 5.15 shows the speed-up of the icoFoam using the two versions of hybrid linear solvers over the Cufflink CG execution of icoFoam. It provides better performance when increasing number of processors, which is noted as P in the figure. The hybrid solvers provides better speed-up than the others in the case of icoFoam.

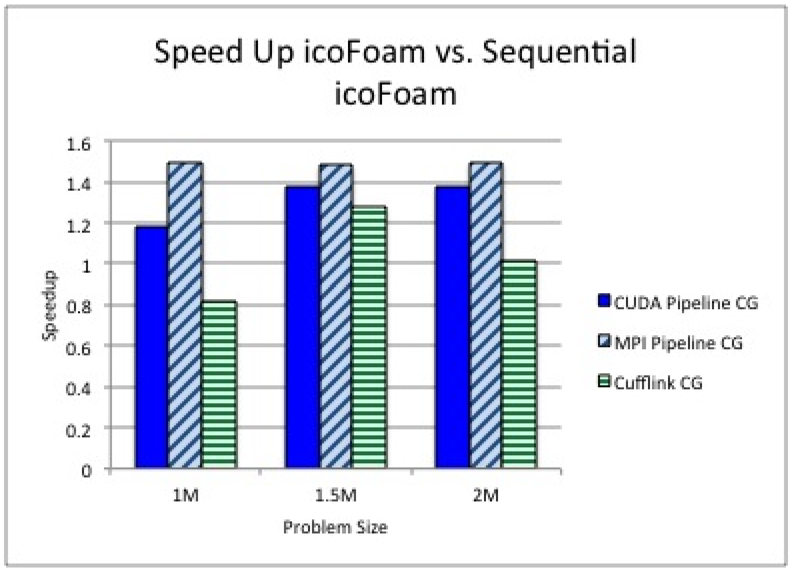


Figure 5.14: Speed-up of the execution time of different parallel run over the sequential execution of icoFoam

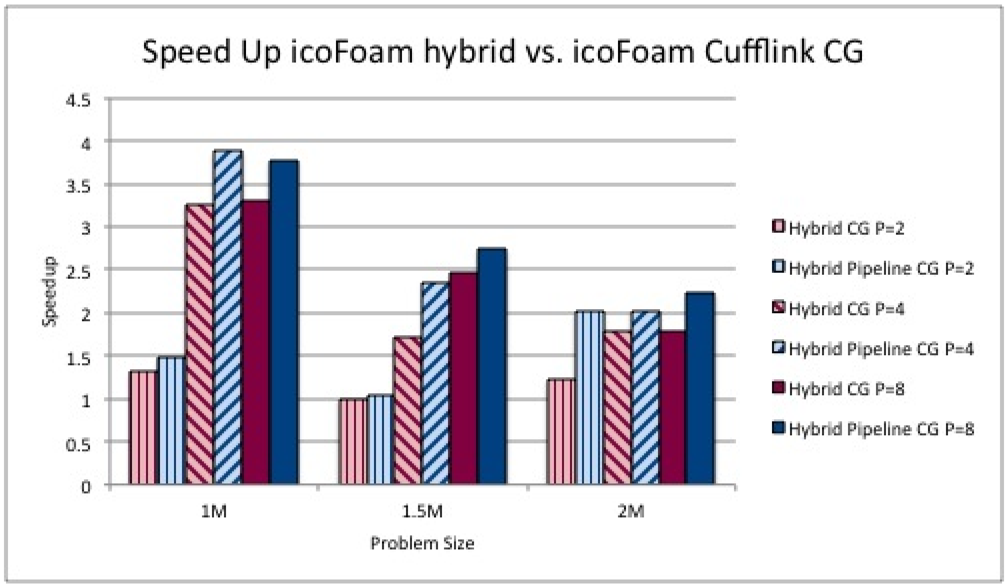


Figure 5.15: Speed-up of the execution time of different parallel run over the icoFoam calling Cufflink CG

## 5.4 Summary

This chapter evaluates the suggested optimizations. The experiments carried out on two local platforms, i.e., Tesla machine and HCL cluster. The heterogeneous decomposition provides a consistent speed-up. Moreover, the pipeline CG CUDA implementation outperforms Cufflink CG, as well as the pipeline CG MPI implementation has better performance than OpenFOAM Parallel CG. Both MPI Pipeline Conjugate Gradient and CUDA Pipeline Conjugate Gradient are called by icoFoam to solve the pressure, and by laplacianFoam to solve the temperature on different mesh size.

We evaluated both the hybrid CG and hybrid pipeline CG solvers using array of problem sizes and test cases. The first one is the impact of the heterogeneous decomposition on both hybrid solvers, and it shows consistent performance gain around 25%. The second one is to compare it with the OpenFOAM Parallel CG on Tesla architecture allocating all the resources i.e. 24 cores and 2 GPU devices. The hybrid solvers overcome the parallel OpenFOAM CG. The third and final test is to show the improvement of the whole package hybrid solvers and heterogeneous decomposition on the total execution time of the selected solvers: icoFoam and laplacianFoam. This optimizations provide good speed-up, however, it should be considered as a stepping stone towards better hybrid heterogenous solvers. The following chapter list the future work and concludes this thesis.

# Chapter 6

# Concluding Remarks

This thesis proposes optimizations aimed at better utilization of the modern computing platforms. We picked OpenFOAM as a CFD package, and studied two solvers as our research target: icoFoam, which is an incompressible flow solver, and, laplacianFoam, which solves the Laplace equation for a passive scalar. The parallelization of OpenFOAM is based on process-level parallelization using MPI. It scales well on homogenous systems, but executes at an inadequate percentage of per-processor performance without special attention to the architecture layout, which is complex, hybrid and heterogenous. Therefore, this approach is bound to expend large amount of hardware resources. On the other hand, Cufflink uses the GPU to accelerate the linear solvers, but, it is bound to number of GPU devices available in the platform. In other words, Cufflink does not support hybrid computations. From the benchmarking result of Cufflink and OpenFOAM using icoFoam solver, the gap between the CPU performance and GPU during the end-to-end computations is not that significant. Therefore, the allocated processors (CPUs) in the platform is good enough to be included in the computation side by side with accelerator.

According to the analysis of the selected solvers, the most time consuming phase is the sparse linear algebraic kernel. Therefore, a hybrid parallel Conjugate Gradient (CG) linear solver has been designed and implemented. In highly heterogeneous architecture, we deploy multi-level parallelism, which uses MPI between each compute nodes and accelerates the sparse linear algebraic kernels in CUDA architectures asynchronously. A load-balancing step is applied by using heterogeneous domain decomposition method, which decomposes the computations in proportion to the speeds of each computing node and takes into account minimizing the communications. The heterogenous decomposition combines the performance model and METIS/SCOTCH libraries. In addition, we designed and implemented the pipeline conjugate gradient solver as an algorithmic improvement, which aims at minimizing the collective communications, and accelerate it using the hybrid MPI+CUDA as well as CUDA and MPI only implementations.

The experimental results show around four times speed-up of laplacianFoam and icoFoam. Also, the pipeline conjugate gradient implementation on multi-GPU outperforms the CUDA conjugate gradient, which is supported by Cufflink library.

## 6.1 Future Research Work

This work can be seen as a stepping stone towards optimized hybrid heterogenous OpenFOAM-based applications. The work presented in this thesis can be extended in the following directions. First, applying the hybrid model to a preconditioned conjugate gradient solver. Second, implementing the hybrid model on other linear solvers, such as algebraic multi-grid solver. Third, employing a dynamic load-balancing algorithm, which adaptively balances the workload during the run-time, by memory-aware work stealing.

# REFERENCES

[1] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, “A tensorial approach to computational continuum mechanics using object-oriented techniques,” *Computers in Physics*, vol. 12, no. 6, pp. 620 – 631, Nov 1998.

[2] (2013) OpenFOAM. http://www.openfoam.com/.

[3] P. Ghysels and W. Vanroose, “Hiding global synchronization latency in the preconditioned conjugate gradient algorithm,” *Parallel Computing*, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819113000719

[4] D. P. Combest and J. Day, “Cufflink: a library for linking numerical methods based on cuda c/c++ with openfoam.” 2011, version 0.1.0. [Online]. Available: http://cufflink-library.googlecode.com

[5] N. Bell and M. Garland, “CUSP: Generic parallel algorithms for sparse matrix and graph computations,” 2010, version 0.1.0. [Online]. Available: http://cusp-library.googlecode.com

[6] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, “PETSc Web page,” 2013, http://www.mcs.anl.gov/petsc.

[7] J. R. Stewart and H. C. Edwards, “A framework approach for developing parallel adaptive multiphysics applications,” *Finite Elem. Anal. Des.*, vol. 40, no. 12, pp. 1599–1617, 2004. [Online]. Available: http://dx.doi.org/10.1016/j.finel.2003.10.006

[8] W. Bangerth, T. Heister, G. Kanschat, L. Heltai, M. Kronbichler, M. Maier, B. Turcksin, and T. Young, “Deal.II: A finite element differential equations,”

http://www.dealii.org/.

[9] H. Meuer, E. Stonhmaier, J. Dongarra, and H. Simon, "Top 500 supercomputer sites," 2013. [Online]. Available: http://www.top500.org

[10] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR factorization on a multicore node enhanced with multiple GPU accelerators," in *Proceedings of IPDPS 2011*, Anchorage, AK, October 2010.

[11] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators," in *High Performance Computing for Computational Science VECPAR 2010*, ser. Lecture Notes in Computer Science, J. Palma, M. Dayd, O. Marques, and J. Lopes, Eds. Springer Berlin Heidelberg, 2011, vol. 6449, pp. 93–101. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19328-6_11

[12] L. Buatois, G. Caumon, and B. Levy, "Concurrent number cruncher: a GPU implementation of a general sparse linear solver," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 24, no. 3, pp. 205–223, 2009.

[13] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," in *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3. ACM, 2003, pp. 917–924.

[14] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser, "A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 2010, pp. 583–592.

[15] A. Cevahir, A. Nukada, and S. Matsuoka, "High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning," *Computer Science-Research and Development*, vol. 25, no. 1-2, pp. 83–91, 2010.

[16] Y. Liu, X. Liu, and E. Wu, "Real-time 3D fluid simulation on GPU with complex obstacles," in *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on.* IEEE, 2004, pp. 247–256.

[17] T. Tomczak, K. Zadarnowska, Z. Koza, M. Matyka, and Â. Mirosław, "Complete PISO and SIMPLE solvers on Graphics Processing Units," *arXiv preprint arXiv:1207.1571*, 2012.

[18] S. Tarsa, T.-H. Lin, and H. Kung, "Performance gains in conjugate gradient computation with linearly connected gpu multiprocessors," *USENIX HotPar*, vol. 12, 2012.

[19] Z. Jamshidi and F. Khunjush, "Optimization of OpenFOAM's linear solvers on emerging multi-core platforms," in *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on*, 2011, pp. 824–829.

[20] A. Kalinov and A. Lastovetsky, "Heterogeneous Distribution of Computations While Solving Linear Algebra Problems on Networks of Heterogeneous Computers," in *Proceedings of the 7th International Conference on High Performance Computing and Networking Europe (HPCN'99)*, ser. Lecture Notes in Computer Science, vol. 1593, Springer. Springer, 1999, pp. 191–200.

[21] A. Kalinov and A. Lastovetsky, "Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 520–535, 2001.

[22] R. Reddy and A. Lastovetsky, "HeteroMPI + ScaLAPACK: Towards a ScaLAPACK (Dense Linear Solvers) on Heterogeneous Networks of Computers," in *Proceedings of the 13th IEEE International Conference on High Performance Computing (HiPC 2006)*, ser. Lecture Notes in Computer Science, vol. 4297, Springer. Bangalore, India: Springer, 18-21 Dec 2006 2006, pp. 242–253.

[23] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker, "ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '96. Washington, DC, USA: IEEE Computer Society, 1996. [Online]. Available: http://dx.doi.org/10.1145/369028.369038

[24] A. Lastovetsky and R. Reddy, "HeteroMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 197–220, 2006.

[25] A. DeFlumere, A. Lastovetsky, and B. Becker, "Partitioning for Parallel Matrix-Matrix Multiplication with Heterogeneous Processors: The Optimal Solution," in *21st International Heterogeneity in Computing Workshop (HCW 2012)*, IEEE Computer Society. Shanghai, China: IEEE Computer Society, May 21, 2012 2012.

[26] J. Dongarra and A. L. Lastovetsky, *High performance heterogeneous computing.* John Wiley & Sons, 2009, vol. 78.

[27] A. Lastovetsky and J. Twamley, "Towards a Realistic Performance Model for Networks of Heterogeneous Computers," in *Proceedings of IFIP TC5 Workshop, World Computer Congress, August 22-27 2004, Toulouse, France*, ser. High Performance Computational Science and Engineering, Springer. Springer, 2005, pp. 39–58.

[28] A. Lastovetsky, R. Reddy, and R. Higgins, "Building the Functional Performance Model of a Processor," in *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006)*, ACM. Dijon, France: ACM, April 23-27 2006 2006.

[29] A. Lastovetsky and R. Reddy, "Data Partitioning with a Realistic Performance Model of Networks of Heterogeneous Computers with Task Size Limits," in *Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04)*, IEEE Computer Society Press. Cork, Ireland: IEEE Computer Society Press, 5-7 July 2004 2004, pp. 133–140.

[30] A. Lastovetsky and R. Reddy, "Data Partitioning with a Functional Performance Model of Heterogeneous Processors," *International Journal of High Performance Computing Applications*, vol. 21, pp. 76–90, 2007.

[31] A. Lastovetsky and R. Reddy, "Data Partitioning for Multiprocessors with Memory Heterogeneity and Memory Constraints," *Scientific Programming*, vol. 13, pp. 93–112, 2005.

[32] A. Lastovetsky and R. Reddy, "Data distribution for dense factorization on computers with memory heterogeneity," *Parallel Computing*, vol. 33, pp. 757–779, Dec 2007.

[33] A. Lastovetsky and R. Reddy, "Two-dimensional Matrix Partitioning for Parallel Computing on Heterogeneous Processors Based on their Functional Performance Models," in *7th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2009)*, Lecture Notes in Computer Science, vol. 6043, Springer. Delft, Netherlands: Lecture Notes in Computer Science, vol. 6043, Springer, September 2009, pp. 112–121.

[34] A. Lastovetsky and R. Reddy, "Distributed Data Partitioning for Heterogeneous Processors Based on Partial Estimation of their Functional Performance Models," in *7th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2009)*, Lecture Notes in Computer Science, vol. 6043, Springer. Delft, Netherlands: Lecture Notes in Computer Science, vol. 6043, Springer, 25/9/2009 2010, pp. 91–101.

[35] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic Load Balancing of Parallel Computational Iterative Routines on Highly Heterogeneous HPC Platforms," *Parallel Processing Letters*, vol. 21, pp. 195–217, Jun 2011.

[36] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data Partitioning on Heterogeneous Multicore Platforms," in *2011 IEEE International Conference on Cluster Computing (Cluster 2011)*, IEEE Computer Society. Austin, Texas, USA: IEEE Computer Society, Sept 26-30 2011, pp. 580–584.

[37] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications," in *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*, Beijing, China, 24-28 September 2012, pp. 191–199.

[38] V. Rychkov, D. Clarke, and A. Lastovetsky, “Using Multidimensional Solvers for Optimal Data Partitioning on Dedicated Heterogeneous HPC Platforms ,” in *Proceedings of the 11th International Conference on Parallel Computing Technologies (PaCT-2011), LNCS 6873*, Springer. Kazan, Russia: Springer, September 19-23 2011, pp. 332–346.

[39] D. Clarke, A. Lastovetsky, and V. Rychkov, “Column-Based Matrix Partitioning for Parallel Matrix Multiplication on Heterogeneous Processors Based on Functional Performance Models,” in *9th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar’2011)*, Lecture Notes in Computer Science 7155, Springer. Bordeaux, France: Lecture Notes in Computer Science 7155, Springer, August 29, 2011 2012, pp. 450–459.

[40] D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa, “Hierarchical Partitioning Algorithm for Scientific Computing on Highly Heterogeneous CPU + GPU Clusters,” in *18th International European Conference on Parallel and Distributed Computing (Euro-Par 2012)*, Lecture Notes in Computer Science 7484, Springer. Rhodes Island, Greece: Lecture Notes in Computer Science 7484, Springer, 27-31 August 2012, pp. 489–501.

[41] B. Chetverushkin, N. Churbanova, A. Lastovetsky, and M. Trapeznikova, “Parallel simulation of oil extraction on heterogeneous networks of computers,” in *Proceedings of the 1998 Conference on Simulation Methods and Applications (CSMA’98)*, Society for Computer Simulation. Orlando, Florida, USA: Society for Computer Simulation, November 1-3 1998, pp. 53–59.

[42] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, “High-performance parallel implicit CFD,” *Parallel Computing*, vol. 27, no. 4, pp. 337–362, 2001.

[43] Y. Liu, “Hybrid parallel computation of OpenFOAM solver on multi-core cluster systems,” p. 90, 2011.

[44] P. Dagnaa and J. Hertzerb, “Evaluation of Multi-threaded OpenFOAM Hybridization for Massively Parallel Architectures,” Tech. Rep. [Online]. Available: http://www.prace-ri.eu/IMG/pdf/wp98.pdf

[45] J. Jgerskpper and C. Simmendinger, "A Novel Shared-Memory Thread-Pool Implementation for Hybrid Parallel CFD Solvers," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, 2011, vol. 6853, pp. 182–193. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23397-5_18

[46] G. Karypis and V. Kumar, "MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.0," http://www.cs.umn.edu/~metis, University of Minnesota, Minneapolis, MN, 2013.

[47] C. Chevalier and F. Pellegrini, "PT-Scotch: A Tool for Efficient Parallel Graph Ordering," *Parallel Comput.*, vol. 34, no. 6-8, pp. 318–331, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2007.12.001

[48] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, "A new era in scientific computing: Domain decomposition methods in hybrid CPUGPU architectures," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 1316, pp. 1490 – 1508, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045782511000235

[49] C. Augonnet, S. Thibault, and R. Namyst, "StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines," INRIA, Rapport de recherche RR-7240, Mar. 2010. [Online]. Available: http://hal.inria.fr/inria-00467677

[50] 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*, vol. 23, no. 2, pp. 187–198, 2011.

[51] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *Innovative Parallel Computing (InPar), 2012*, 2012, pp. 1–12.

[52] J. H. Ferziger and M. Perić, *Computational methods for fluid dynamics.* Springer Berlin, 1996, vol. 3.

[53] M. Beaudoin and H. Jasak, "Development of a generalized grid interface for turbomachinery simulations with OpenFOAM," in *Open Source CFD International Conference. Berlin, Germany*, 2008, pp. 4–5.

[54] J. D. Anderson, *Computational fluid dynamics.* McGraw-Hill New York, 1995.

[55] H. Jasak, "Error analysis and estimation for the finite volume method with applications to fluid flows," Ph.D. dissertation, Imperial College London (University of London), 1996.

[56] F. H. Harlow and J. E. Welch, "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface," *Physics of fluids*, vol. 8, p. 2182, 1965.

[57] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Pittsburgh, PA, USA, Tech. Rep., 1994.

[58] D. Clarke, A. Ilic, A. Lastovetsky, V. Rychkov, L. Sousa, and Z. Zhong, *Design and optimization of scientific applications for highly heterogeneous and hierarchical HPC platforms using functional computation performance models*, ser. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2013.

[59] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.3.0. [Online]. Available: http://www.meganewtons.com/

[60] D. Clarke, Z. Zhong, V. Rychkov, and A. Lastovetsky, "Fupermod: A framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous hpc platforms," in *Parallel Computing Technologies.* Springer, 2013, pp. 182–196.

[61] (2013) HCL cluster. http://hcl.ucd.ie/wiki/index.php/HCL_cluster.

[62] (2013) Paraver: Performance Analysis Tools. http://www.bsc.es/computer-sciences/performance-tools/paraver.

[63] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a generic framework for managing hard-

ware affinities in HPC applications," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on.* IEEE, 2010, pp. 180–186.

[64] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, vol. 44, 1995, pp. 17–31.

[65] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[66] K. Fürlinger and D. Skinner, "Capturing and visualizing event flow graphs of mpi applications," in *Euro-Par 2009–Parallel Processing Workshops.* Springer, 2010, pp. 218–227.

[67] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach.* Gulf Professional Publishing, 1999.

[68] C. Xavier and S. Iyengar, *Introduction to Parallel Algorithms*, ser. A Wiley interscience publication. Wiley, 1998. [Online]. Available: http://books.google.ie/books?id=W3Ld65MnwgkC

[69] C. D. Polychronopoulos, "Parallel programming and compilers," 1988.

[70] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: the complete reference.* MIT press, 1995.

[71] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck, "GPGPU: General-purpose computation on graphics hardware," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188672

[72] A. Abdelfattah, J. Dongarra, D. Keyes, and H. Ltaief, "Optimizing memory-bound numerical kernels on gpu hardware accelerators," July 2012.

[73] R. Farber, *CUDA application design and development.* Access Online via Elsevier, 2011.

[74] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier, 2012.

[75] A. Lastovetsky, "Heterogeneity in parallel and distributed computing," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 1523–1524, 2013.

[76] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-passing Interface*, ser. Scientific and engineering computation. MIT Press, 1999, no. v. 1. [Online]. Available: http://books.google.ie/books?id=xpBZ0RyRb-oC

[77] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "OpenMPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* Springer, 2004, pp. 97–104.

[78] I. Buck, "GPU computing with Nvidia CUDA," in *SIGGRAPH*, vol. 7, 2007, p. 6.

[79] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of parallel and distributed computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[80] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.

[81] C.-T. Yang, C.-L. Huang, and C.-F. Lin, "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters," *Computer Physics Communications*, vol. 182, no. 1, pp. 266–269, 2011.

[82] A. Lastovetsky, R. Reddy, V. Rychkov, and D. Clarke, "Design and implementation of self-adaptable parallel algorithms for scientific computing on highly heterogeneous hpc platforms," no. arXiv:1109.3074, 09/2011 2011.

[83] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on platforms with memory heterogeneity," in *Europar 2010 / Heteropar'2010*, Lecture Notes in Computer Science 6586, Springer. Ischia-Naples, Italy: Lecture Notes in Computer Science 6586, Springer, 31/08/2010 2011, pp. 41–50.

[84] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on GPUs using compile-time and run-time strategies," *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[85] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Tech. Rep., 2008.

[86] Y. Saad, *Iterative Methods for Sparse Linear Systems: Second Edition*. Society for Industrial and Applied Mathematics, 2003. [Online]. Available: http://books.google.ie/books?id=Uoe7xBOhS5AC

[87] D. E. Keyes, L. C. McInnes, C. Woodward, W. D. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, X. Jiao, K. Jordan, D. Kaushik, E. Kaxiras, A. Koniges, K. Lee, A. Lott, Q. Lu, J. Magerlein, R. Maxwell, M. McCourt, M. Mehl, R. Pawlowski, A. Peters, D. Reynolds, B. Riviere, U. Rüde, T. Scheibe, J. Shadid, B. Sheehan, M. Shephard, A. Siegel, B. Smith, X. Tang, C. Wilson, and B. Wohlmuth, "Multiphysics simulations: Challenges and opportunities," *International Journal of High Performance Computing Applications*, vol. 27, pp. 4–83, 2013.

[88] W. D. Gropp and D. E. Keyes, "Domain decomposition on parallel computers," *Impact Comput. Sci. Eng.*, vol. 1, pp. 421–439, 1989.