OPTIMIZATION OF FLUID SOLVERS WITH RESPECT TO FAULT TOLERANCE AND MEMORY LATENCY

by

Atis Degro

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Physics

Committee:

| | |
|---|---|
| _____ | Dr. Rainald Löhner, Committee Chair |
| _____ | Dr. Fernando Camelli, Committee Member |
| _____ | Dr. Juan R. Cebral, Committee Member |
| _____ | Dr. Chi Yang, Committee Member |
| _____ | Dr. Paul So, Department Chair |
| _____ | Dr. Donna M. Fox, Associate Dean, Office of Student Affairs & Special Programs, College of Science |
| _____ | Dr. Ali Andalibi, Dean, College of Science |

Date: _____ Spring Semester 2020
George Mason University
Fairfax, VA

Optimization of Fluid Solvers with Respect to Fault Tolerance and Memory Latency

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Atis Degro
Master of Science
Technical University of Munich, 2012
Professional Master Degree
Riga Technical University, 2010

Director: Dr. Rainald Löhner, Distinguished Professor
Department of Physics and Astronomy

Spring Semester 2020
George Mason University
Fairfax, VA

# Dedication

I dedicate this dissertation to my beloved parents and family.

# Acknowledgments

First, I would like to thank my dissertation advisor Dr. Rainald Löhner for giving me the opportunity to pursue a doctoral degree while working under his guidance. His knowledge, enthusiasm and dedication have helped and motivated me throughout my years at George Mason University.

Additionally, I would like to thank Dr. Fernando Camelli for his passionate teaching, for always finding time to give me advice and the frequent friendly conversations.

I would like to thank Dr. Chi Yang for her assistance throughout my doctoral studies. Her attention to the details of this new program consistently guided me in the right direction.

I would like to thank Jayshree Sarma, Director of the Office of Research Computing at George Mason, as well as her team for all the support I have received. Work done on the ARGO cluster would not have been possible without their help.

I would like to thank my colleague and friend Alejandro Figueroa. From the beginning of this program, he has provided advice and camaraderie, and without him this whole program would have been very different.

Last but not least, I would like to express my deepest gratitude to my parents, my sisters, the rest of my family and my friends. They have inspired me and shown me incredible support throughout my doctoral studies.

# Table of Contents

# List of Tables

# List of Figures

# Abstract

OPTIMIZATION OF FLUID SOLVERS WITH RESPECT TO FAULT TOLERANCE AND MEMORY LATENCY

Atis Degro, PhD

George Mason University, 2020

Dissertation Director: Dr. Rainald Löhner

Constant advancement of computational systems lifts the theoretical boundaries of what is possible to achieve with numerical simulations. In order to fully utilize the capabilities of advanced computational resources, codes must be adapted accordingly.

One major challenge that comes with petascale and exascale computing is fault tolerance. The larger the number of nodes used for code execution the lower the expected time between hardware failures. Based on available research data, several failures per day can occur when running massively parallel applications. Several fault tolerance enabling techniques have been analyzed and proposed in past years; however, currently there are no fault tolerant computational fluid dynamics (CFD) solvers that can efficiently execute an application at the Exascale or Petascale level. The aim of this PhD dissertation is to analyze and implement available resilience techniques to develop a fault-tolerant CFD solver.

The second challenge addressed in this work is the memory latency problem for CFD codes. Many CFD codes that exhibit low computational intensity (flops per RAM access) 'saturate' the memory bandwidth of modern chips after only a few cores; therefore, any possible benefits of utilizing more of the available cores are minimized. While previously the CPU speed determined how fast a certain code could be executed, currently, the memory

access speed sets the upper limit for the solver's performance. That is the reason why some fluid solvers can achieve only 10-15 percent of the peak performance of the floating point pipelines on recent CPU cores. This has led to the development of minimal memory access loop (MMAL) options for finite difference solvers. Several loops are described and analyzed.

Finally another approach to address the memory latency problem for CFD codes is investigated. Intrinsic instructions in C++ are used to code the subroutine that obtains the right hand side (RHS) for the finite difference approximation. Intrinsic instructions take advantage of the full vector length and maximize the number of operations that can be done simultaneously.

# Chapter 1: Introduction

## 1.1 Problem statement

In 2008, petascale supercomputing was achieved. The IBM-built supercomputer named Roadrunner went online and exceeded 1 petaflops on the Linpack test [1]. The number of Petascale computers on the Top500 supercomputer list grew steadily. Since June 2019, all the top 500 supercomputers perform at petascale – at least 1.022 petaflops [2]. The progress, however, does not stop at petascale. Planning for exascale began before the first petascale computer went online. Furthermore, the first prognosis anticipated an exascale supercomputer by the year 2015 [3]. The current prognosis anticipates the first exascale supercomputer to go online in 2021 [4].

Exascale computing holds the potential to lift the current limits of scientific computing and enable new possibilities. However, exascale computing comes with new challenges. In order to fully utilize the exascale performance, several issues have to be addressed.

One main issue which has been repeatedly discussed regarding exascale computing, is resilience [5]. Today's supercomputers suffer daily from hardware failures. The mean time between failures (MTBF), depending on the supercomputer, can be as frequent as every 7 hours [6]. Due to the increased number of components expected in an exascale computer, the MTBF will likely reduce even further – potentially down to minutes. Applications that follow a master/slave hierarchy of work assignment do not suffer significant computational losses when a process fails. New replacement processes can be spawned at minimum expense without loss of valuable data. Meanwhile, fault tolerance is a more serious issue in field solvers where processors are working in a flat hierarchy. Each process works simultaneously on a sub-domain of the problem and critical information between them is exchanged each

time-step. In this case, loss of a process implies loss of information that is necessary to advance the execution.

Although work on fault tolerant computing has been ongoing for decades, currently there are no production codes that can deal with node or core failures. Furthermore, most of the advancements made in fault tolerant computing are not applicable when using Fortran programming language. Due to the differences in programming language capabilities, strategies suggested for C++ can not be used in Fortran. This work concentrates on investigating the available fault tolerant techniques and designing a fault tolerant field solver written in Fortran.

Another issue addressed in this work is the increasing gap between memory and CPU performance. Both CPU and memory transfer performances have been steadily increasing. CPU performance, however, is increasing at a much higher rate. This has led to a situation where data transfer from different levels of memory can not 'feed' the CPU fast enough to enable the theoretical computational performance. Applications where such a scenario is observed are memory bound – peak performance is limited by the system bandwidth between CPU and memory. One way of addressing this issue is to design algorithms that reduce the necessary amount of data transfer. This work investigates two possible approaches to reduce the memory latency:

- rearranging execution order so that the maximum amount of data could be reused after it has been loaded in higher level memory (L1, L2 cache); and,
- using intrinsic instructions in order to utilize full vector length of the registers. This way more operations can be executed simultaneously on larger data sets, thereby increasing the performance.

## 1.2 FDFLO

Fault-tolerance strategies as well as memory latency reduction attempts were designed and tested using FDFLO [7–10]. FDFLO is a finite difference code that solves the quasi-incompressible (artificial compressibility) Navier-Stokes equations and is based on the following set of building blocks:

- Cartesian spatial discretization;
- Embedded or immersed boundaries for complex geometries;
- Explicit timestepping via low-storage Runge-Kutta schemes;
- Conservative formulation for advection and divergence;
- Easy extensions to high-order stencils;
- Ordered access to memory;
- Long 1-D loops (for optimal vector, OMP and GPU performance);
- Use of halo points to impose boundary conditions.

FDFLO has been in development for a number of years and has been used for both fundamental research of turbulent phenomena as well as industrial large-eddy simulations (LES) of complete car configurations.

## 1.3 Thesis outline

The body of this dissertation is organized around the issues highlighted in the above problem statement.

Chapter 2 is dedicated to fault tolerant fluid solver development. First, a more thorough motivation for necessity for a fault tolerant solver is provided followed by an overview of currently available techniques and related work. Second, the developed and implemented

strategies are presented and discussed. Finally, the results of several test cases and overall performance evaluation are reported.

The attempts to minimize memory access using minimal memory access loops (MMALs) comprise chapter 3. This includes the motivation and detailed description of MMALs potential advantage over traditional approach with respect to data transfer from memory. Six different minimal memory access loop options are introduced and implemented in the finite difference solver FDFLO. Test runs of the implemented loops are conducted on different hardware systems and several problem sizes are tested. Chapter 3 concludes with the result discussion and outlook.

Chapter 4 is dedicated to the use of intrinsic instructions as a tool for improving the performance of fluid solver. Challenges and limitations of intrinsic instructions are discussed. Intrinsic calls are used to implement an alternative subroutine that obtains the RHS values. Both single precision and double precision options are explored. The subroutines are tested and compared with the traditional version of RHS update. To conclude, the suggestions for possible future work are given.

Finally, chapter 5 presents an overall summary of efforts towards optimization of fluid solvers with respect to fault tolerance and memory latency. The main outcomes are evaluated. Further directions of future work are identified.

# Chapter 2: Fault Tolerant Fluid Solver

## 2.1 Introduction

The current path to exascale computing foresees tens of thousands of heavily populated nodes (i.e. millions of cores) working on the same time-critical problem. One of the emerging issues with millions of cores is the time between failures. Random failures of cores or the communication between cores - commonly referred to as 'faults' - that are expected to occur every couple of minutes pose a serious problem for production runs that need hours or days to complete. This problem has not emerged so far because:

- for most machines, the number of (high quality) cores allowed for a single run is still in the range of tens of thousands [11, 12], i.e. times between failures occur only every few hours; and,
- due to their Message Passing Interface (MPI) implementations, most computing centers do not allow for fault-tolerant computing, i.e. if any MPI process/core fails, the run terminates immediately.

It should come as no surprise then that none of the production codes currently in place can deal with cores/nodes failing. To date, the approach has been to periodically write all restart information to disk (e.g. every hour), so that if the machine experiences a malfunction, only the last hour of computing is lost. This approach requires constant human supervision or elaborate restart scripts, so that a considerable number of productive hours are not be lost should a node or core fail.

By default, each error encountered when using MPI is fatal and the simulation is aborted. Since the MPI-2 standard, it is possible to change the default behavior and allow the

execution of the application to continue even after an error has been encountered. This feature has been used by many when trying to develop a fault-tolerant MPI implementation. The new capabilities offered by fault-tolerant MPI implementations have motivated research in resilience strategies and fault-tolerant application development.

From a fault-tolerant perspective, the simplest implementations are those that follow a master/slave arrangement. Typical applications are distributed searches [13], or embarrassingly parallel problems such as parameter scoping or evolutionary optimization [14]. For these cases, the loss of a processor is immaterial, as a new process can be spawned to replace it without any detriment to the overall application. Fault tolerance becomes much more difficult for field solvers where processors are working simultaneously on different subdomains, and critical information is exchanged between them every iteration or time step. In this case, the failure of one processor leads to an irrecoverable loss of information.

## 2.2 MTBF

Back in 1965 Moore's law predicted that every 2 years the number of components per integrated circuit will double [15]. One can see how correct this prediction has been in Figure 2.1. This constant progress affects also the overall trend and performance of the supercomputers. The list of the top 500 supercomputers is updated twice each year and shows the statistics of the current computers with the highest peak performance [16]. Over the time the supercomputers on this list have shown a steady increase in size with respect to the number of compute nodes and CPUs (Figure 2.2). This progress towards ever bigger computers is driven by the demand from science and industry. We are currently reaching the level of exascale in terms of the floating point operations per second.

Increasing number of components on the computer lead to decreasing failure rates. Mean time between failures (MTBF) is expected to go as low as several minutes. Researchers have addressed the issue of hardware failures on supercomputers ever since the first talks about petascale computing [18]. The predictions for petascale computing with respect to failure rates turned out to be quite accurate, now predictions for exascale computing are being

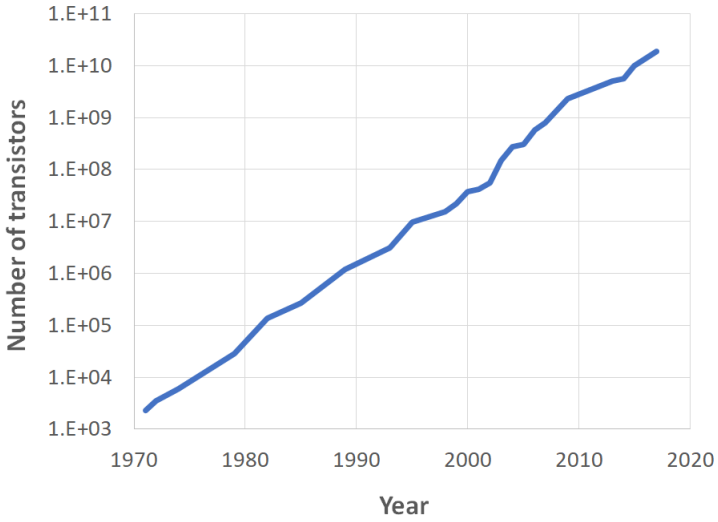



Figure 2.1: Increase in numbers of transistors that fit onto a microprocessor over the years. Data taken from [17].

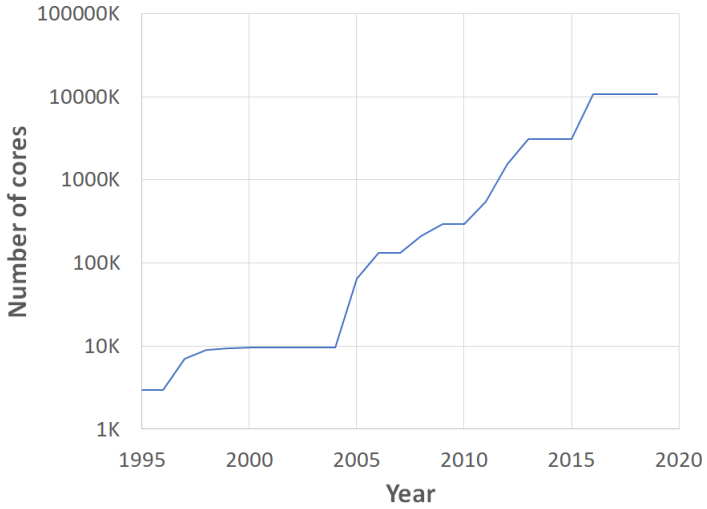



Figure 2.2: Increase in the maximum number of cores in the top 500 supercomputers

Table 2.1: Supercomputer statistics regarding MTBF

| Name of the supercomputer | Number of nodes | Number of cores | MTBF [h] |
|---|---|---|---|
| Jaguar XT4 | 7832 | 31328 | 36.91 |
| Jaguar XT5 | 18688 | 149504 | 22.67 |
| Jaguar XK6 | 18688 | 298592 | 8.93 |
| Eos | 736 | 23553 | 189.04 |
| Titan | 18688 | 560640 | 14.51 |
| CEA Tera100 | 4300 | 140000 | 20 |
| IBM Blue Gene | 40960 | 163840 | 180 |
| Blue Waters | 22640 | 724480 | 7 |

made. Several studies and surveys analyzing data from supercomputing centers have been carried out over the past years to come up with a good prediction model to estimate the MTBF on a given machine [6, 19–26]

Main issues addressed in these studies are:

- which components are more prone to failure;
- how does MTBF change with respect to the life cycle of machine's components;
- does MTBF depend on system type;
- does new generation hardware increase or decrease the reliability;

Most common type of failure ( 45%) is failure of a single node or CPU [19]. There is no clear agreement whether the MTBF correlates better with number of CPUs, number of nodes or number of sockets. Most likely it is the combination of all. The overall trend however is clear. Figure 2.3 shows the correlation between MTBF and the number of CPUs. The data points used in this chart are taken from several sources [6, 20, 22, 27] and are presented in Table 2.1.

In Figure 2.3 the red curve denotes an exponential trendline. The number of data points is small due to the lack of available information of failure statistics of different supercomputers. The deviation of the data points from the trendline comes from the complexity of the issue, as mentioned above, the MTBF depends on a combination of different factors.

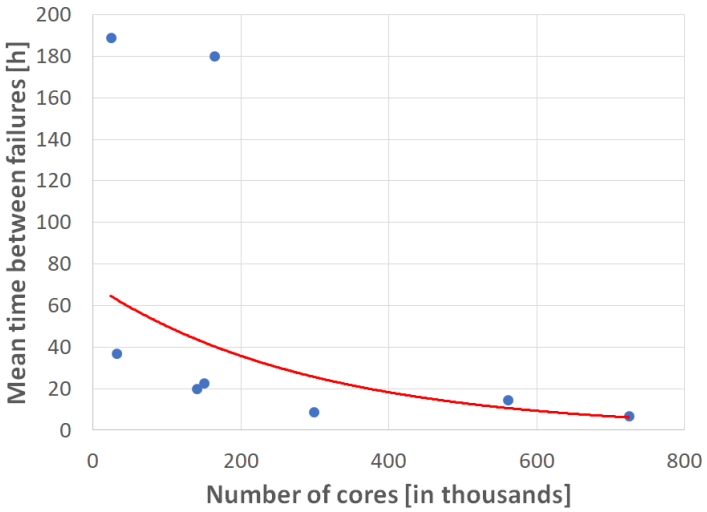



Figure 2.3: Mean time between failure depending on the total number of cores

For example, the two supercomputers with high MTBF (Eos and IBM Blue Gene) clearly stand out from the overall trend. IBM Blue Gene has only 4 CPUs per node, Eos has a low overall number of nodes. Although these are not unique characteristics, they possibly contribute to the high MTBF. The trendline is not perfect and there are data points that seem to not follow it at all. The general pattern of the correlation between MTBF and the number of CPUs however is clear. As the number of CPUs per supercomputer rises, the MTBF is going to reduce.

Based on the findings of [26], supercomputers encounter failure temporal locality. That means, if a failure occurs, it is very likely that the next failure will occur after a time that is notably lower than the MTBF. Furthermore, if the expected runtime of a simulation is larger than the MTBF, there is a very high chance that several and not just one failure will be encountered.

According to [22] the MTBF varies significantly over time. The MTBF of the same supercomputer can change as much as 4 times in the period of one year. This fact even further adds to the unpredictability of failure occurrences.

The data shown in this section does not prove that a failure will occur during a specific

application execution. It does show, however, how likely and unpredictable failures are and illustrates the significance of fault-tolerant applications for high performance computing.

## 2.3 Current situation regarding fault tolerant applications

In this section the current situation regarding fault-tolerant applications is discussed. A brief overview of available tools and applications is given.

### 2.3.1 MPI library functionality with respect to fault tolerance

Work on the MPI began in 1991 and the first MPI standard, the MPI 1.0 was released in 1994. Since then, MPI has been the state of the art shared memory communication library for parallel programs.

By default each encountered error when using MPI is a fatal error and the simulation is aborted. The reason for this is to ensure a correct calculation. With the hardware advancement the supercomputers got bigger, more and more nodes(processors) were available for a single simulation. With the increasing number of the processors decreased the average time between element failure. This was the motivation for research on development of resilient computational methods. However the MPI standard by default remains non-fault tolerant. According to the MPI-3 standard: "An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an MPI exception." [28]

The MPI standard does however provide some tools for possible fault handling. Several predefined error handlers are available in MPI:

- **MPI_errors_are_fatal** The handler, when called, causes the program to abort on all executing processes. This has the same effect as if **MPI_abort** was called by the process that invoked the handler [28]. This is the default error handler that is

assigned to every initialized MPI communicator.

- **MPI_errors_return** The handler has no effect other than returning the error code to the user[28]. What happens after an error is encountered depends on the specific implementation. There is no guarantee that the user will be able to keep using the communicator once an error is encountered. This option does however give user a chance to store any data or execute other non MPI calls before the simulation is aborted.

Additionally to the pre-defined error handlers, the user has the option to write a custom error handler function. The custom error handler can then be assigned to a MPI communicator. In this scenario the custom error handler function is going to be called by each MPI process that encounters and error (MPI call does not return MPI_success). It will be shown in the next chapter how this functionality is used for development of fault tolerant MPI extensions.

### 2.3.2 MPI fault tolerant extensions

A lot of work has been done on development of MPI extensions to increase the fault tolerant options provided by standard MPI. In this chapter introduction to some of the most known MPI extensions is provided by giving a short summary of the methodology, functionality and background.

#### MPI-FT

MPI-FT is a fault tolerant version of MPI [29]. The proposed methodology consists of a mechanism to detect the failures and a recovery procedure. The detection and recovery is done by a centralized process called Observer that monitors the application. Once a failure is detected the Observer notifies the alive processes. That is followed by the recovery procedure. There are two proposed solutions for the recovery procedure. For the first method distributed buffering of the message process is performed on each process. After the failure the Observer recovers all buffered the messages from the alive processes and

resends them to the replacement processes that have substituted the dead processes. For the second method the Observer receives and stores all the message traffic during runtime. After a failure is detected the Observer, similarly to method one, resends all the messages issued for the dead processes to the replacement processes.

### FT-MPI

FT-MPI is a fault tolerant version of MPI [30]. The first significant difference from the MPI standard is that the default error handler is MPI_errors_return. FT-MPI is built on the HARNESS (Heterogeneous Adaptive Reconfigurable Networked SyStem), an experimental meta-computing system [31]. FT-MPI is based on the MPI-1.2, includes some parts of the MPI-2 and extends some of the semantics of MPI to make recovery from failed processes possible [32]. FT-MPI provides four different error modes also called 'communicator modes' that can be specified at the beginning of the application. The four modes are as follows [33]:

- **ABORT** - corresponds to the standard MPI default and aborts on an error;
- **BLANK** - after a failure MPI_COMM_WORLD doesn't change size, the failed processes are neither removed or replaced, alive processes keep their original ranks as assigned before the crash;
- **SHRINK** - similar to blank, the failed processes are not replaced, however, they are removed, leaving no 'holes' in the communicator. The communicator is shrunk to the size of remaining alive processes. The remaining processes might have a new rank; and,
- **REBUILD** - this is the default mode of the FT-MPI. After the failure the communicator is first shrunk and then new processes are spawned to replace the failed processes. The size of the MPI_COMM_WORLD doesn't change and neither do the ranks of the processes.

Although the FT-MPI provided good fault tolerant functionality it was never implemented in the MPI standard and has been discontinued. Due to the lack of generality (tied to a specific computing system) it was never widely adapted or implemented in any production codes.

**FA-MPI**

FA-MPI is a Fault-Aware version of MPI [34]. It provides extensions to the MPI standard thereby providing options for implementing fault tolerant methods in the applications. FA-MPI is restricted to non-blocking MPI communication. This can be seen as a disadvantage when considering implementing fault tolerance in legacy codes. FA-MPI uses TryBlock API extensions to introduce transactional behavior. Series of operations are firstly "tried" to execute and are "committed" only when all the operations have succeeded. The operations are "rolled-backward" or "rolled-forward" if some of the operations fail. The advantage of FA-MPI is that it doesn't deal exclusively with process failure. FA-MPI also claims to have smaller overhead since the failure is not detected/mitigated/isolated/recovered per operation but per group of operations [34, 35].

**ULFM**

User Level Failure Mitigation can be described as an attempt to introduce a standardized fault tolerant extension of MPI that would be implemented in the MPI standard. It has several similarities with the FT-MPI. The MPI Forum's Fault Tolerance Working Group has been working on implementing standard fault tolerant MPI by adding additional calls to the existing MPI standard [36]. The main additions to the existing MPI consists of supplementary error codes and five supplementary interfaces. The following is a brief description of these supplementary MPI semantics. Supplementary error codes [37]:

- MPIX_ERR_PROC_FAILED – an MPI call will return this error if failure of a process on the communicator is preventing the completion of the MPI operation.

- MPIX_ERR_PROC_FAILED_PENDING – an MPI call will return this error when a potential sender matching a non-blocking wildcard source receive has failed.
- MPIX_ERR_REVOKED – an MPI call will return this error if either of the ranks in the application has initiated the revoke operation on the communicator.

Supplementary MPI functions [37]:

- MPIX_Comm_failure_ack – this is a local operation that acknowledges all locally notified failures on the communicator. After this call unmatched receive operations that would otherwise have raised an error will proceed without raising further errors since the error is already acknowledged.
- MPIX_Comm_failure_get_acked – this is a local call that returns the group of acknowledged failed processes on the communicator. This call should be called after MPIX_Comm_failure_ack.
- MPIX_Comm_revoke – this function is used to revoke a communicator. It notifies all the processes associated with this communicator that it is revoked. It is not a collective function therefore there are no matching calls on remote processes. It is a local call with global effect. After a communicator is revoked all non-local calls in this communicator will invoke an error of class mpix_err_revoked.
- MPIX_Comm_shrink – this functions creates a new communicator. On the input is the communicator with failed processes and on the output a new communicator that excludes all the failed processes. It is a collective call and has to be called by all the alive processes.
- MPIX_Comm_agree – this is a collective communication call ensuring that all the alive processes agree on a bitwise 'AND' operation. Using this call after MPIX_Comm_failure_ack lets users synchronize the knowledge of failures across the communicator.

ULFM added functions provide tools to deal with overall process of encountering process failure in following phases:

- notification phase – ensuring that all the processes are aware of the failure;
- propagation phase – stopping all the activity of the affected communicator by revoking it; and,
- recovery phase – removing the failed processes from the application and getting consensus between all the surviving processes about the state of the recovered (healthy) communicator before proceeding with the execution of the application.

ULFM makes it possible to implement local or global recovery depending on the type of the application. ULFM is currently one of the most widely used fault tolerant MPI extensions.

**MPICH**

MPICH is a high-performance widely portable implementation of the MPI standard since MPI-1 [38]. MPICH has added to the implementation the following fault tolerant calls from ULFM:

- failure_ack, failure_get_acked;
- MPIX_Comm_shrink;
- MPIX_Comm_agree;
- MPIX_Comm_revoke.

In order to use these calls, however, MPICH has to be configured before installation with the following flag:

- –enable-error-checking=all

In addition, users must enable a runtime flag each time they execute the application:

- –disable-auto-cleanup

This flag prevents the process manager to kill processes if any of the processes experience failure [39]. During the course of this work, it was found that the MPICH version of the MPIX_Comm_shrink call is limited to 32 MPI processes. That is not the case when using ULFM. MPICH is widely used which makes it a good and simple option for trying and implementing basic fault tolerance methods. It seems, however, that the support for fault tolerance in MPICH could be discontinued [40].

**Fenix**

Fenix is a fault tolerant application programming interface (API). It uses ULFM provided MPI extension semantics. Fenix has two main functionalities - repairing communicators and restoring state of the application from a checkpoint [41].

Fenix recovers resilient communicators that are initialized using `Fenix_Init`. The repair process supports both shrinking and non-shrinking recovery of the failure affected communicator. In order to use the non-shrinking recovery option additional redundant resources (spare ranks) have to be assigned at the beginning of execution of the application. Fenix automatically captures errors that result from MPI operations on the "fenix" communicators.

For data recovery Fenix provides a designated data storage API. The user, however, can choose to use other data recovery options. The Fenix provided data recovery API relies on storing application variables and arrays to redundant data storage [41].

The recovery process restores the application to a defined recovery point as opposed to the caller which makes The Fenix API not compatible with Fortran codes [42].

**Reinit**

Reinit is a global-exception, roll-back recovery model. Reinit was developed as an attempt to address the shortcomings of ULFM. Contrary to ULFM where broken communicators are repaired, Reinit relies on reinitialization of the MPI. Once a fail-stop error is detected,

MPI reinitializes and restarts the application at a previously defined restart point. In doing so, MPI is also restarting any failed ranks. Reinit depends on very accurate (only failed processes are reported as failed) and synchronized (all the alive processes are eventually informed about the failed processes) fault detector in the MPI runtime [43].

### 2.3.3 Related work

A few of the efforts reported to date in this field are mentioned in what follows.

- A fault-tolerant implementation of an application solving 2D partial differential equations (PDEs) by means of a sparse grid combination technique has been reported by Ali et al. [44]. It is capable of surviving multiple process failures caused by the faults. This implementation uses the capabilities of the User Level Failure Mitigation (ULFM) extension of MPI [44]. After a detection of a failure the communicator is revoked, then shrunk to the remaining processes and lastly new processes are spawned to replace the failed ones. Three different data recovery procedures including checkpoint/restart have been tested. The faults are injected sending (SIGKILL) to the processes at a certain point from within the code. Faults are being detected by calling `mpi_barrier` on the Parent communicator. This application is developed using C++.

- The theory for fault tolerant multi-level Monte Carlo (FT-MLMC) for solving the two-dimensional stochastic Euler equations of gas dynamics has been developed and tested by Pauli et al. [45]. This is a work on developing a fault tolerant ALSVID-UQ (Multilevel Monte Carlo Finite Volume solver for uncertainty quantification in hyperbolic systems of conservation laws [46]. Fault tolerance is implemented using the ULFM MPI extension. After encountering a failure the communicator is revoked and shrunk to the remaining (alive) processes. The recovered communicator is of reduced size. Faults are introduced by killing processes using the system exit call. Faults are introduced at random times based on Weibull distribution model. Faults are being detected by calling `mpi_barrier` on the parent communicator. This application is

developed using C++.

- The Fenix MPI Fault Tolerance software library compatible with the Message Passing Interface (MPI) to support fault recovery without application shutdown has been used in conjunction with PDE solvers by van der Wijngaart et al. [47]. The developers report the successful implementation of Fenix MPI (mentioned earlier in this chapter) in the S3D code for the numerical simulation of combustion [48].

- A number of publications have reported on efforts to make basic matrix operations and linear equations solvers fault-tolerant [49–51]. Contrary to the works mentioned above these efforts discuss resilience to soft errors. The fault tolerance is achieved at the algorithm level and does not use additional MPI calls.

It can be noticed here that none of the related works are using Fortran as the programming language. A considerable number of production codes are still written in Fortran, and given the large number of man-years that were required to write and debug them, probably will not be rewritten. The challenges when using Fortran will be discussed in the following chapter.

### 2.3.4 Fault tolerance when using Fortran

It can be seen in the previous section that fault tolerance has been considered an issue already for several years. Significant amount of work has been done on MPI extensions to achieve fault tolerance in advanced numerical computations. The currently available MPI extensions have been implemented in numerical solvers and tested. The results so far look promising. Today, there is no standard for developing fault tolerant applications, however, the work is ongoing and the future seems promising. There is, however, an important fact that should be addressed. All the related example applications from the previous section are written in C++. Furthermore, most of the MPI fault-tolerant extensions described in the previous section do not support Fortran as the programming language.

Fortran, being one of the oldest programming languages, is still also one of the most used

ones. Especially for scientific applications, even though it has not seen much development and change throughout it's existence. And there is a reason for that: Fortran can generate a very fast native code, one of the reasons why it is ideal for scientific computing. It is also highly optimized for vectorization and therefore good for supercomputer applications. Fortran is readable and understandable. But it is not just the fact that Fortran in many cases is the preferable programming language for designing new scientific applications. There are many legacy codes written in Fortran. These are large codes that have been used for a long time and are capable of taking advantage of increasing performance of the supercomputers. Porting such a code to C++ in order to ensure fault tolerance is not reasonable. This makes a strong argument for the necessity of a fault tolerant option for Fortran codes.

Reasons why fault tolerance is harder to achieve in Fortran with the currently available options has been thoroughly discussed and analyzed [42].One of the main reasons is described in more detail in what follows.

Many of the currently available fault tolerant solutions propose the following recovery to ensure fault resilience, once a failure has been detected, the surviving processes automatically return to a previously specified location in the code. This could be anywhere in the code but usually in the main function and before beginning of the outermost loop. Once the surviving processes arrive at this location, application can be repaired and then resumed.

C++ semantics provide a non-local control option to use during recovery. This option in C++ is **longjmp** which can be seen as a non-local *goto*. The use of this call can simplify the fault-tolerant functionality implementation. **Longjmp** synchronously and directly diverts all the remaining processes to the repair instance of the code from wherever the fault has been encountered. With this call the control is transferred to a "jump point" which is previously defined in the code with a call to **setjmp**. As said before, this is a part of the suggested recovery procedure. In case of a failure all MPI processes can resume execution at a consistent location. In most cases that is after a successful roll-back recovery.

In Fortran there does not exist a **setjmp/longjmp** equivalent. GOTO statements in Fortran are limited within the scope of the current subroutine (procedure). That means

that other strategies and methods have to be developed to implement fault tolerance in a Fortran code. ULFM was chosen as the main MPI fault tolerant extension to achieve a fault resistant version of the FDFLO code. The developed methods are explained in detail in the following sections of this work.

## 2.4 Levels of Fault Tolerance

Fault tolerance is the ability of the application to overcome errors encountered during the execution. One application can be resilient to certain types of hardware failures while failing if a software error occurs. Another application can resist both software and hardware errors. Both applications can be considered fault tolerant but there is clearly a difference. When talking about fault tolerance different levels and types of fault tolerance may be identified. The two extremes being:

- aborting after each encountered error no matter the error type (no fault-tolerance); and,
- 'surviving' any type of faults and any number of faults (complete fault-tolerance).

While it might be impossible to achieve complete fault-tolerance, in between these two extremes there are many possible scenarios. A code can be fault tolerant with a certain level of confidence with respect to a certain type of error depending on how likely it is for the computation to fail/abort in case such an error is encountered. There are several factors that set the theoretically achievable upper limit of fault tolerance with respect to such errors. To name a few:

- Failure of Rank 0. When using MPI, rank 0 is considered the master process. It carries unique information about the MPI communicator. There are currently no procedures in place to recover a communicator if the rank 0 fails. Considering the total amount of processes being used, the chances that exactly rank 0 is going to fail are rather low.

- Non-application related software error. For example, the system environment errors of the supercomputer, workload manager or other tools.

This work only investigates and deals with hardware errors. More specifically, failures of individual CPUs and nodes. Some of the factors that influence the level of fault tolerance are:

- How many copies for rollback recovery are kept in memory during runtime. The recovery information is used after a failure has occurred. If one copy of the rollback information is kept in memory, the recovery is only possible if the failure has not affected both, the process itself and it's recovery information simultaneously. The more copies of the rollback recovery are kept in memory the higher the chances that recovery will be possible.

- How many spare resources are used for the simulation. Spare resources are used to replace the failed processes. The number of assigned spare resources sets the limit on how many process failures can be overcome.

- Whether a code is able to survive all failure scenarios including the worst cases or only the most common ones. Failure can occur at any point of the execution process. There might be parts of the application that are error resistant while others are not. The ratio between the vulnerable parts of the application and the resilient parts of the application is a strong indicator of the level of fault tolerance.

To find the most cost effective option, it is important to look at the likelihood of each of the failure scenarios, and the associated overhead for fault tolerance involved. At present, with few fault-tolerant codes in operation, the information available is insufficient in order to know the optimal solution. This work focuses on investigating and addressing these issues.

## 2.5 Fault Tolerant Code Design

As stated previously, the focus of this work is to design a fault-tolerant code written in Fortran. There are many legacy codes written in Fortran used in scientific computing. These codes can take advantage of ever increasing performance of supercomputers. However, these codes are not fault tolerant since at the time of their development MTBF was not a concern. The aim of this work is to develop a methodology for implementing fault tolerance in Fortran applications. Furthermore, how it can be achieved with simple modifications that do not demand large changes made to the existing code.

In order to obtain fault-tolerant codes, methodologies need to be developed to:

- discern which nodes/cores have failed;
- store the information required to restore a previous state;
- restore the state of the run before failure; and,
- re-assign work to either the remaining working processors or a set of 'reserve' processors used to handle failing cores/nodes.

In the following, we treat each of these aspects in turn. The overall approach followed here attempts to minimize the changes required to large-scale codes when enabling fault-tolerant computing.

## 2.6 Detecting and discerning node/core failure

One of the most frustrating user-experiences is the 'churning' of runs where cores/nodes have failed. The user thinks the code is running fine, any run-diagnostics such as `qstat` or similar commands shows time being spent, but in fact the run has stopped, waiting for information to come from or arrive at the node that has failed. Assuming the worst-case scenario that a number of nodes may have failed, one needs to develop methods to discern

which nodes have failed that do not need to have the consent or messaging from all nodes (as, for example an `all_reduce` operation) in order to detect failure. As the aim was to achieve fault-tolerant computing with the least amount of changes to existing production codes, a simple call to:

```
mpi_barrier(MPI_COMM_WORLD,ierro)
```

was added to several locations in the code. If all processors are alive, `ierro=0`, otherwise an error has occurred. Additionally the `mpi_barrier` calls add synchronization to the code which, as was observed, increases the resilience to the failures. Since it is not possible in Fortran to execute a global *goto*, it is important that all the processes obtain information about a failure in a fairly synchronous manner (at the same location in the code). This is important to ensure that all the alive processes go through the same call sequence during recovery and do not get stuck at other parts of the code.

The addition of an `mpi_barrier`-call adds a CPU time of 5-25 $\mu sec$ [52]. Given that for large-scale runs a single time step or iteration requires at least 3 orders of magnitude more time this is considered insignificant.

## 2.7 Storage of restart/recovery information

As any node may fail, the information that is needed in order to use either another (spare/reserve) node to continue the run or to be sent to the remaining working nodes must be stored outside the node. The simplest (and often used) way is to write to disk and then continue. This is extremely time-consuming: writing a 500 Mels restart file to disk may take as much as 1-2 minutes (!). The reason is that most large-scale machines have separated compute nodes and disk storage. When trying to store a complete restart all compute nodes are writing simultaneously to disk, creating a bottleneck. Specialized hardware and software have been reported considerable improvements [53], but an informal survey of colleagues worldwide confirmed the figure reported above. Therefore, the idea is to store the restart information in scratch arrays that are saved in other processors. At the very least, two

copies of this restart information are required: one that stays on the processor (in case it does not fail), and one that is stored on another processor (local and buddy checkpointing). This other processor should be as far away as possible on the network in order to minimize the probability of an unrecoverable state due to multiple simultaneous neighboring failures. Clearly, more copies could be stored across the processors in order to prevent scenarios such as failure while storing data or simultaneous failure of a node and the node that keeps its recovery information, but two copies seem to be a very efficient way to proceed.

The amount of data required for recovery from failure depends heavily on the field solver, the time-marching scheme employed, and the physics modeled. For computational fluid dynamics (CFD) codes on stationary grids, the main data items required are element connectivity, boundary conditions, coordinates and unknowns at the points/elements. If grids are moving, the mesh velocity may have to be stored as well. For rheologically complex fluids material history variables are required. Furthermore, for time marching schemes that require the information from several previous time steps (e.g. Adams-Bashforth, Adams-Moulton or implicit Runge-Kutta schemes) this information needs to be stored for all required time steps. Computational structural dynamics (CSD) codes may have to store additionally plasticity, material or damage history data at Gauss-points, original strain deformation tensors, and other quantities, i.e. a much larger amount of data for the same gridsize. Furthermore, the usual memory vs. run-time tradeoffs are also encountered here: in some cases, the construction of the additional information required to run a field solver (e.g. for CFD solvers the edges of the mesh, geometry factors, distance to wall, overlapping grid information, etc.) may take CPU time, so storing them for restart/recovery could be advantageous.

In order to reduce memory requirements only the minimum amount of restart/recovery information required was kept. Furthermore, in order to improve the code's transparency and extendability, the restart/recovery information was stored in the same way as restart files would be written to and read from disk. For the CFD codes used here, this implied integer and real backup arrays for control, body, diagnostics, grid generation, domain and field diagnostics data.

## 2.8 Recovery from failure

After an error in the MPI communicator is encountered, the first thing is to ensure that all the remaining processes are aware of the state of the communicator. Since in Fortran no global *goto* is available, the next step is to make all the remaining processes return to the main function without getting stuck somewhere in the code due to the failed communicator. This is achieved by implementing additional checks in different levels of the code that skip calculation steps if the MPI communicator has encountered an error. Once in the main loop, all the processes from a damaged communicator are directed to the recovery call sequence. The recovery sequence is responsible for identifying all nodes that have ceased to operate, as well as the processors that have their backup information. If the information to restore a pre-failure state is not available (either because many nodes have failed, a network outage has occurred or any other catastrophic machine failure has taken place) the run stops and is restarted with the last restart state written to disk. If, on the other hand, the information to restore a pre-failure state is available, all nodes restart from the last saved state. After the recovery a check is performed to ensure that all the processes have restarted to the same time step.

## 2.9 Re-assignment of work

With the assumption that the information to continue the run from a given backup time step is available, three options are viable:

1. Load rebalancing: in this option, one utilizes the remaining working nodes, rebalancing the load before continuing the run. This is more involved, as one needs to combine information from nodes that have ceased to work with information of working nodes, and load balancing for complex physics is a non-trivial task [11, 54].

2. Spawn new processes to replace the processes of failed nodes: after the failure the

application determines how many nodes (processes) have failed and spawns the corresponding number of new ranks to replace the failed entities. In this case when submitting a job on a cluster one has to make sure that: a) The environment allows to spawn new processes and b) There are extra nodes reserved for the execution.

3. Reserve nodes that were not used before: the key idea here is to allocate at the beginning of the run a small number of so-called 'reserve nodes' that can be used to replace the nodes that fail. Contrary to the spawning option, the job is started on all the nodes from the beginning. However, the processes of the reserve nodes are left idle until a failure occurs. If a failure occurs, one of the reserve nodes is assigned to the list of 'running' or 'active nodes' and the MPI communication (or 'MPI universe') table is modified to reflect this change. Note that at the code level, no change is needed for the tables and indices of the information that is sent between processors.

We consider this a very elegant solution, as the information to restore a run and continue is straightforward to implement with any production code. The number of reserve nodes required will depend on the mean time to failure of the machine being employed. It is estimated that this number is very small compared to the total number of nodes (and remains at a relatively constant percentage of the number of nodes required for a run), so that the extra burden in resources is insignificant.

The main subroutine used to renumber the processes and replace the communicators is listed in Appendix A.1.

## 2.10 Methods to introduce failure

To test a fault-tolerant code under real life scenarios, one would have to test it under real supercomputer hardware failures. Although failures on supercomputers are common and frequent, they are also random. Causing a realistic failure on a supercomputer, however, can be harmful for the system itself therefore cannot be used as a testing strategy. The aim of any large-scale computing resource or computing center is to maximize utilization and

operating time. Therefore, it is not easy to test fault-tolerant codes on typical production systems.

Several methods for introducing failure were used in order to test the effectiveness of the implemented fault-tolerant procedures. The methods explored can be listed in ascending level of complexity as follows:

- Sending a kill signal from within the code (e.g. at the end of a randomly selected time step). This method is the furthest away from a realistic node failure. It can, however, be successfully used to check whether the implemented recovery strategies are functional. The disadvantage of this method is that the failure occurs only at specific places of the code (where it has been implemented) and cannot account for the randomness of the realistic failure process.

- Killing processes externally from the terminal using '`kill -9 PID`'. This method addresses the disadvantage of the previous method. Since the processes are killed externally from the application, the failure can happen at any time during the execution, at any part of the code. To ensure reliable testing results, high number of tests have to be conducted. This was achieved by executing a script that repeatedly goes through the following steps:

  - starting the simulation;
  - waiting for the problem to initialize and reach past the first checkpoint;
  - obtain the IDs of the related processes;
  - randomly selecting and killing one of the processes (either active or spare);
  - waiting for the code to recover and recording the result; and,
  - aborting the simulation and restarting the procedure.

  Although this method does not represent a real node or core failure, it is a practical way of doing numerous tests and observing the code's ability to recover from random core failures.

- Logging into one of the active cluster nodes and killing all simulation related processes. Unlike the first two methods where tests were done on a powerful workstation, this method is used in a cluster environment. Since computing center resources are limited as said above large-scale computing resource aim is to maximize the utilization, the focus of this and the following method was qualitative - perform less but more realistic tests. In this method all the processes related to the application are killed. Although not representing an actual node failure, it does test the code against losing multiple processes (all located on the same node). Test was repeated on several consecutive nodes during the same application execution, replicating a consecutive node failure.

- Rebooting one of the active cluster nodes using 'sudo reboot'. This method is the most realistic representation of an actual node failure. Not only all the processes related to the application execution are killed, but so are all the other underlying software tools responsible for the communication between the cluster nodes. Even though it is the most realistic, it does not represent exactly the scenario of actual node failure. When using the 'sudo reboot' command in the cluster environment any IO operations that are running at that point are actually gracefully terminated, any pending checkpoint file write file will be written and only afterwards the machine will become unavailable.

One way to replicate a realistic node failure would be to physically 'unplug' the node during runtime. However, as mentioned above, such action could harm the hardware and is not available. Already the last method described above demands certain administrator privileges and was used as the final test.

## 2.11 Test cases

The fault-tolerant algorithms described above were implemented in FDFLO. For the cases shown below, the restart/recovery information was stored every 10 time steps, which for the cases run implies every $0.1 - 1.0$ seconds. As this is a cartesian finite difference code using explicit Runge-Kutta time integration, the main storage required is comprised by the

field point arrays (5 unknowns for the flow, 16 for diagnostics), and the boundary, halo and mpi-exchange information. For a case with $10^6$ points (1 Mpts) this implies approximately 200 MBytes.

In order to test the performance of the fault tolerant implementation all previously described methods to introduce failure were applied. In total several thousand runs were performed over a period of several weeks. For the majority of tests a cluster environment was used. This allowed to test the fault tolerance of the code in a 'close to realistic' node failure scenario when one of the computation nodes gets rebooted during runtime. Tests were performed using 4 compute nodes with 8 CPUs on each. The nodes were rebooted at random points in time after the beginning of the calculation and at different intervals. Some of the tests were also conducted on larger machines with up to 32 nodes and 1,280 cores. All tests were carried by using only the CPUs on these machines.

### 2.11.1 Taylor-Green Vortex

This is an example that is often used in the large eddy simulation (LES) and direct simulation of Navier-Stokes (DNS) literature [55, 56]. The domain spans $[-1 \leq x, y, z \leq 1]$ with periodic boundary conditions in each dimension. The initial conditions for the velocities $u, v, w$ in $x, y, z$ were set to:

$$u = sin(\phi x)cos(\phi y)cos(\phi z) \quad , \quad v = -cos(\phi x)sin(\phi y)cos(\phi z) \quad , \quad w = 0 \quad . \tag{2.1}$$

The initial conditions quickly deteriorate into smaller vortices, leading to an increase in the dissipation rate. At later times, the laminar viscosity leads to a decrease in velocities.

Figure 2.4 shows the distribution of the absolute value of the velocities in the three principal planes going through the center of the domain $(x, y, z = 0)$ at time $T = 10.0$ for a grid of 10 Mpts using a 4th order finite difference scheme in space and time. The grid had 64 domains, and was run on 8 MPI processes (i.e. 8 domains per MPI process). At

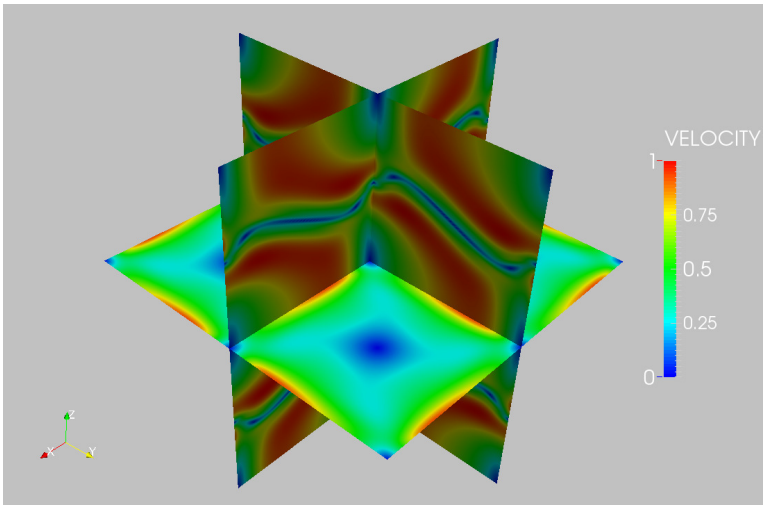



Figure 2.4: Taylor-Green Vortex: Velocity Magnitude in Principal Planes at Time $T = 10.0$

the beginning of the run, an extra 4 MPI processes were allocated. During the runtime at different instances of time processes were terminated by sending '`kill -9 PID`' signal. The recovery rate of this setup was 98% (i.e. the code failed approximately 2 out of every 100 runs, with a sample size of several thousand). Note that as stated before the proposed procedures are not completely fail-safe: if a node fails while restart information if being stored (which in this case was in the range of 2%) a previous state can not be recovered and the code stops.

The same test case was run on a cluster using 4 compute nodes with 8 CPUs on each (16 active MPI processes and 16 spare processes). Nodes were rebooted at random points in time after the beginning of the calculation and at different intervals. The simulation successfully substituted the failed processes and terminated the run without problems.

### 2.11.2 Ahmed Body

The Ahmed body is a widely used testcase in the automotive industry [57,58]. The surface mesh provided consisted of `O(19 Ktria)` and `O(10 Kpts)`. It was run through the FECAD pre-processor, which invoked the PRE-FDFLO grid generator. Two cases were run. The first had `ndomn=705` domains, with a minimum cell size of $dx = 0.0060\ m$ and a total

point count of `npoin=5.75 Mpts`, of which `nactp=4.53 Mpts` were actually updated (some points are not updated are they are in halo regions or within the car). The second had `ndomn=3,993` domains, with a minimum cell size of $dx = 0.0026\ m$ and a total point count of `npoin=36.9 Mpts`, of which `nactp=23.1 Mpts` were updated. The following physical and numerical settings were employed:

1. Density: $\rho = 1.0\ kg/m^3$
2. Velocity: $|\mathbf{v}| = 30\ m/sec$
3. Speed of sound: $c = 150\ m/sec$
4. Laminar viscosity: $\mu = 0.7 \cdot 10^{-5} kg/m/sec$
5. Smallest cell/element size: $h_{min} = 0.0060\ m$ (case 1), $h_{min} = 0.0026\ m$ (case 2)
6. Largest cell/element size: $h_{max} = 0.0480\ m$
7. Spatial discretization: 4th order, central + artificial visosity
8. Temporal integration: explicit 4th order low-storage Runge-Kutta

Figures 2a-d show the overall geometry, the grid, and the instantaneous velocities and Q-criterion for the symmetry plane ($z = 0$). This case was run using 32 MPI processes: 24 active processes and 8 reserve processes. During the runtime at different instances of time processes were terminated by sending the '`kill -9 PID`' signal. The recovery rate of this setup was 94% (i.e. the code failed to continue for approximately 6 out of every 100 runs). The difference in failure rate between this and the previous set of runs is due to the difference in the machines used and the times required for restart storage versus computations (recall that if a node fails while restart information if being stored (which in this case was in the range of 6%) a previous state can not be recovered and the code stops.

This test case was also run on a cluster using 4 compute nodes with 8 CPUs on each (16 active MPI processes and 16 spare processes). Nodes were rebooted at random points

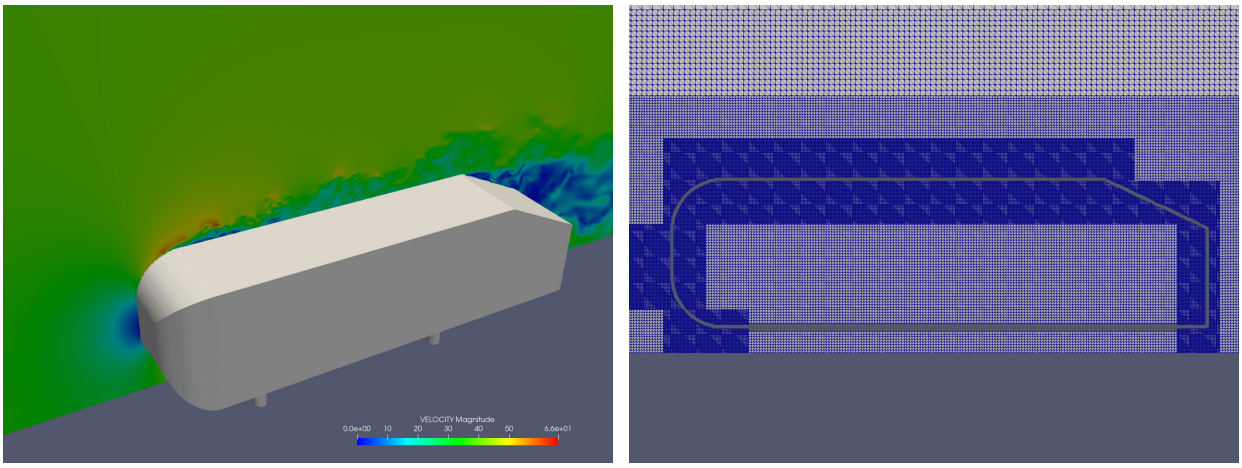



Figure 2.5: Ahmed Body: Geometry and Grid System Employed (Cut Along $z = 0.0$)

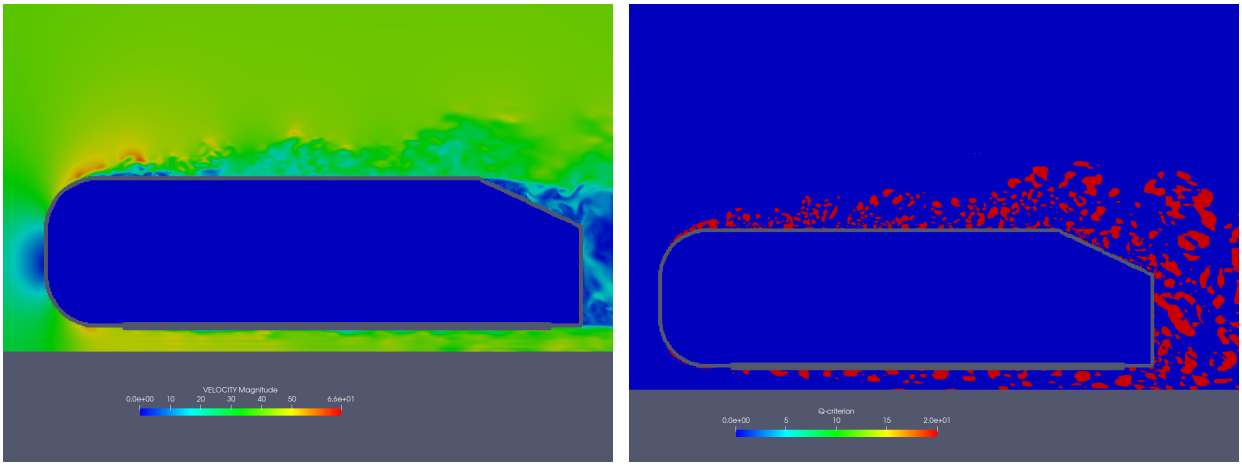



Figure 2.6: Ahmed Body: Velocities and Q-Criterion in Plane $z = 0$

in time after the beginning of calculation and at different intervals. The fault-tolerant procedures developed successfully substituted the failed processes and terminated the runs without problems.

A third series of test runs was conducted on a larger machine using 32 nodes with 40 cores each, i.e. a total of 1,280 cores. As before, the fault-tolerant procedures developed successfully substituted the failed processes and terminated the runs without problems. With several recoveries during the course of the run, the timings observed for these runs (`ndomn=3,993`, `nactp=23.1 Mpts`) were of the order of $T_s = 0.154$ $sec/step$.

The results obtained may be summarized as follows:

- When a process is killed from within the code, the recovery rate is at 100%.
- When processes are killed externally from the terminal using '`kill -9 PID`', the recovery rate is in the range of 94-98%.
- When nodes were rebooted externally, the recovery rate from 2 node (8 MPI processes each) failure while using spare nodes for recovery was 100%. Due to the complexity (manual, queue on the cluster) a smaller amount of tests was run with rebooting the active cluster nodes. It is therefore realistic to assume that the true recovery rate for node failure would approach the 94-98% rate of single process failures.

### 2.11.3 Fault Tolerance Overhead Estimation

Fault tolerance is important, but it is also necessary to know at what cost. In the previous sections, the developed strategies and implementation in a finite difference solver has been explained. The example test cases show good results with respect to fault tolerance – the application is able to survive multiple core or node failures with a high success rate.

As the next step, an answer was sought to the question as to how much additional computational time the fault tolerance implementation does cause. In other words: what is the overhead compared to the non fault tolerant application. The main fault-tolerant extensions made to the code are as follows:

- Synchronization calls with global agree on the state of the "health" of the communicator. At the end of each time step a specific synchronization call sequence has been added. Since some of these calls needs to have a global consensus over the communicator, the execution time can be larger than that of a simple *barrier* call. It was important to identify the magnitude of impact these synchronization calls have on performance. The execution time for different number of MPI processes was recorded. It was also tested if the number of used compute nodes change the execution time.

- `mpi_allreduce` can be a point of failure if not designed with additional calls. Occasionally, if a process would fail while a `mpi_allreduce` call is being executed the application would get stuck. Reason for this is that some of the processes would not notice the error and continue with the execution while others go to the recovery. In order to address this issue two things were changed. First, the `mpi_allreduce` call was substituted with the non-blocking `mpi_iallreduce` call. Second, additional synchronization calls with global agree on the state of the "health" of the communicator were added. The detailed call sequence can be found in Appendix A.2. It is important to note that the amount of these calls per application run depends on the frequency of diagnostic data (point data, surface data, etc) dumps.

- Local and buddy checkpointing (described in section 2.7) between MPI processes is one of the major additions to the code compared to the non fault tolerant application. As mentioned previously it is not a time consuming process. It is, however, important to record how long exactly it takes. The influence of problem size (number of used cores and nodes) was tested as well.

- Recovery process is the combination of all the calls that are being executed from the moment a failure has been detected until successfully resuming the execution of the application with a fixed communicator.

Timings of the recovery process can not be directly compared with the non fault-tolerant application since there is no such process. However the combination of all the three items

can serve as a reference to illustrate the gain of a fault tolerant code. The amount of time spent during the "manual" roll-back recovery (the most common recovery procedure after aborted simulation in case of a hard error) is magnitudes higher.

The results of the above mentioned test have been presented in Table 2.2. Different problem sizes have been tested.

## 2.12 Results

Figure 2.7 to Figure 2.9 show changes in the execution time of fault tolerant call sequences depending on the number of MPI processes and the number of compute nodes. Figure 2.10 shows the comparison of total execution time between the fault tolerant version and non fault tolerant version as a function of the number of MPI processes and the number of compute nodes. These tests were performed using ARGO cluster (George Mason University compute resource) [59].

The results clearly show that the lowest execution time of MPI call sequences is recorded when the application is running on a single node. Also the average time spent on backup and total execution time of the application is the lowest when running on one node. This is an expected result since the communication between the processes is more local. The execution time however is directly correlated with the number of compute nodes. The time for backup for example is the highest when running on 32 MPI processes and 2 compute nodes. Execution time for the `mpi_iallreduce` extended call sequence for the fault tolerant version as well as the `mpi_allreduce` call for the standard version of the application stays almost constant when running on 2 to 8 compute nodes and increases only when running on 16 compute nodes.

One of the main fault tolerant application performance characteristics is the overall overhead of the fault tolerant calls. How much more the total application execution time is increased by the fault tolerant functionality compared to the standard non fault tolerant version. Based on the test performed so far both versions perform at almost the same rate when running on one compute node. For the largest case tested, 256 MPI processes on 16

compute nodes, the overhead was 12%. The largest difference is recorded when running the application with 128 MPI processes on 8 compute nodes - 41%. The results presented here show the execution time only of the more complex MPI call sequences implemented to achieve fault tolerance. In addition, a simple call to:

```
mpi_barrier(MPI_COMM_WORLD,ierro)
```

was added to several locations in the code as described in section 2.6. Some of these calls are executed multiple times per time step. Although execution time of each single call is insignificant, the sum of all the calls lead to the recorded fault-tolerance overhead.

The execution time of separate call sequences as well as the overall application time clearly depends on the number of used compute nodes, however, it is not a linear correlation. It depends also on the underlying compute resource, the type of compute nodes and the connection between the nodes.

Table 2.2: Timings of Faut tolerant call sequences and implementations [Sec]

| | Number of MPI processes and used cluster nodes | | | | |
|---|---|---|---|---|---|
| | 256p/16n | 128p/8n | 64p/4n | 32p/2n | 8p/1n |
| End of time step call sequence | 0.04 | 0.03 | 0.02 | 0.01 | 0.01 |
| Fault tolerant iallreduce | 0.08 | 0.06 | 0.05 | 0.05 | 0.02 |
| Non fault tolerant allreduce | 0.03 | 0.03 | 0.03 | 0.03 | 0.01 |
| Average time for backup | 0.47 | 0.91 | 1.59 | 3.03 | 0.08 |
| Total time (fault tolerant) | 202 | 210 | 196 | 126 | 157 |
| Total time (non fault tolerant) | 180 | 149 | 172 | 101 | 160 |

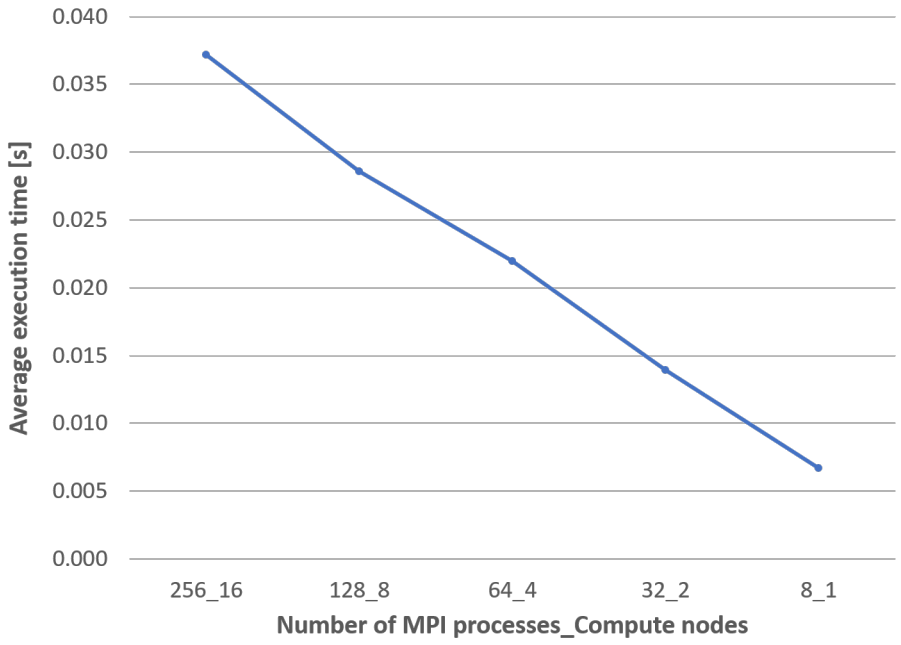



Figure 2.7: Average execution time of the fault tolerant call sequence at the end of the time step

## 2.13 Conclusions and outlook

Fault-tolerant computing options based on the use of restart information stored on and off the compute node, and the use of reserve processes have been developed, implemented and tested in a large-scale, production field solver taken from the CFD domain.

The tests conducted to date have shown good results, with recovery rates in excess of 90% after externally lost processes both on local machines and in cluster environments (i.e. close to realistic node failures).

The proposed fault-tolerant scheme does not cover all possible scenarios. Some of the scenarios not covered include: failure of nodes and/or communication during backup or recovery of backup, failure of node 1 (the master node), and simultaneous failure of a node and the node where its backup information is stored. The probability of these scenarios is low, but should also be considered in the future.

The largest test case run to date was performed using 521 MPI processes. As of now,

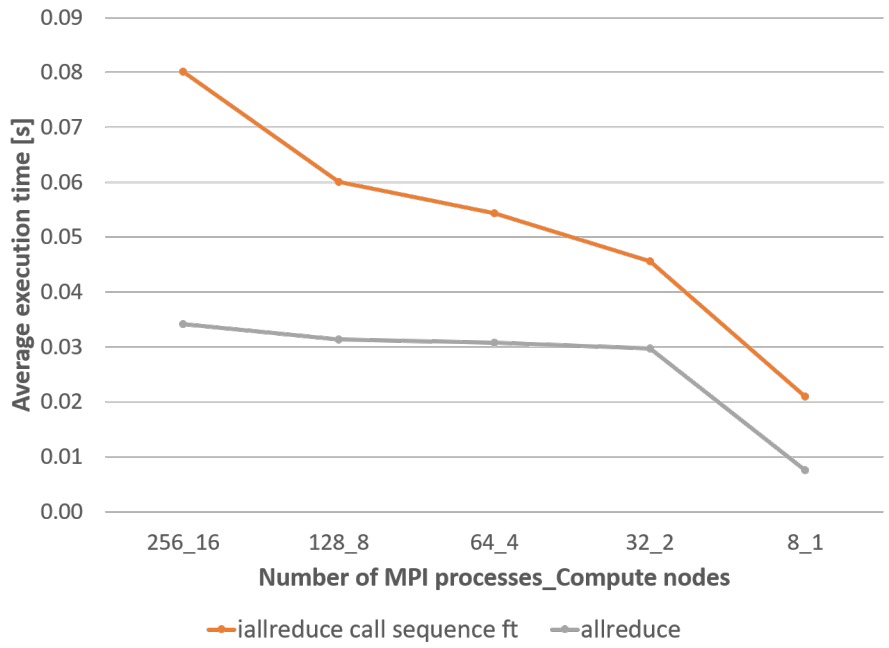



Figure 2.8: Comparison between the execution time of fault tolerant iallreduce call sequence and regular allreduce call

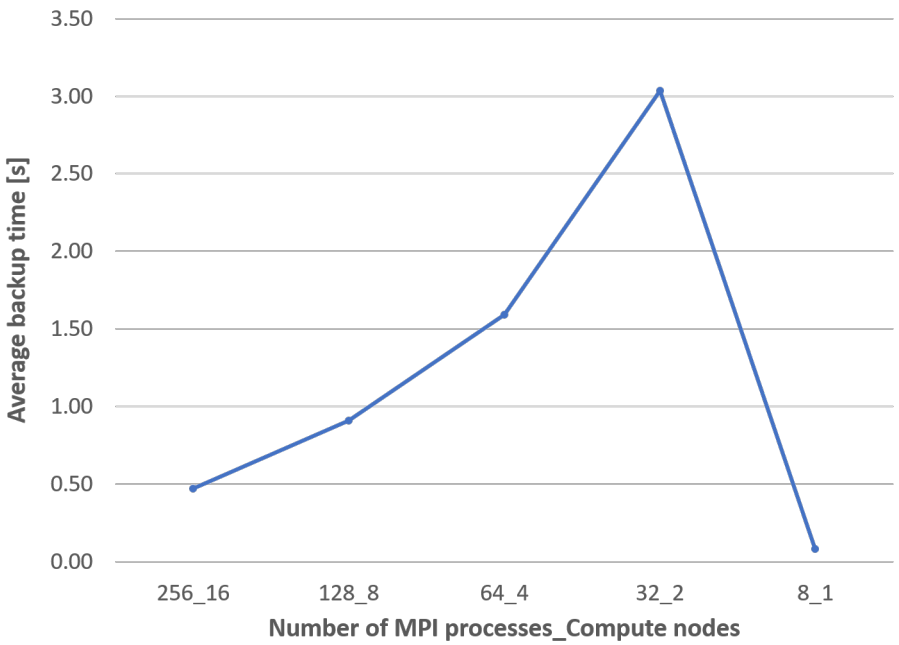



Figure 2.9: Average execution time of information backup between "buddy" processes

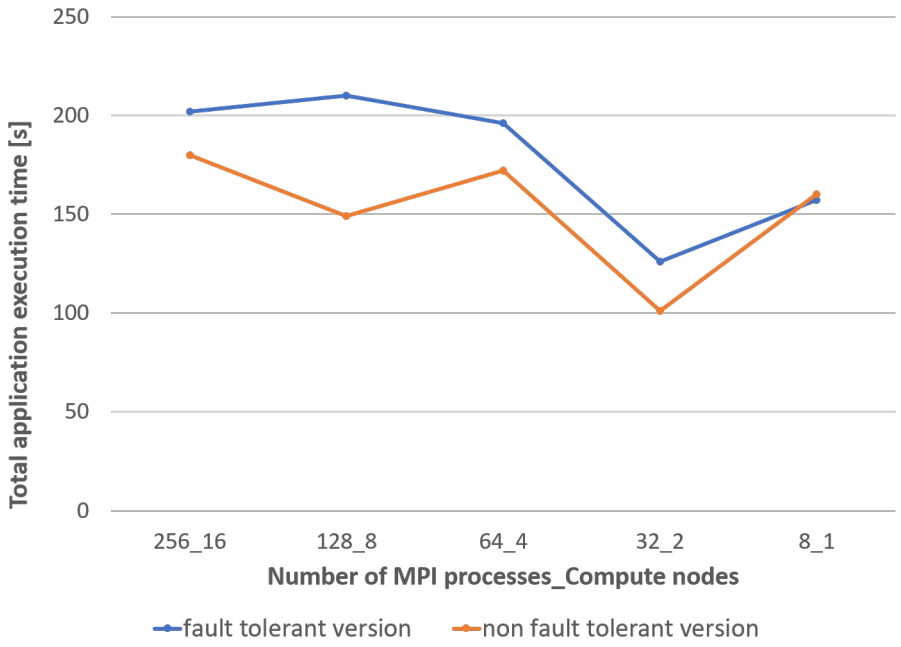



Figure 2.10: Comparison between the total execution time of fault tolerant version and non fault tolerant version

no size limitations have been encountered when using the ULFM MPI library extension.

The computational overhead of the field solvers is very low (explicit time-marching and finite differences). The fault tolerant implementation adds a run-time penalty that is in the range of 6%-41% percent, depending on the spatial and temporal approximation used. The run-time penalty strongly depends on the number of compute nodes that are used for the simulation and the underlying compute resource itself. More compute nodes does not, however, mean larger run-time penalty. The largest case tested (256 MPI processes on 16 compute nodes) yield a run-time penalty of only 12%. Further investigations are underway to reduce this overhead without influencing the very high recovery rate of the code. We remark that this run-time penalty is incurred due to checking for faults, and not for backing up restart/ recovery information. Backing up restart/recovery information every 10 time steps (where in the case of FDFLO the bulk of the information consists of the 4/5 unknowns and 16 diagnostics variables at each gripoint) adds a negligible amount

of CPU requirements. Obviously, the amount of information required for restart/ recovery may be higher for other codes, in which case, if necessary, one could adjust the backup frequency.

Future work should be dedicated to address several issues. One of the main weak points of the current implementation is the run-time penalty. The main runtime overhead comes from the synchronization calls placed in the code in the form of `mpi_barrier(MPI_COMM_WORLD,ierro)`. The effect each of the barrier calls has on the overall recovery rate has to be further investigated in order to find a more optimal solution.

As reported above the run time penalty varies significantly depending on the number of compute nodes used. More tests on different compute resources should be performed to identify the source of the difference. Even larger test cases should be performed in order to test the run-time penalty when running on hundreds of compute nodes.

The code could be further improved by implementing strategies for optimal backup frequency. Several studies have investigated and reported methods for determining an optimal checkpointing period [60–62]. Such strategies could be used to take into consideration the type of the underlying problem when choosing the backup frequency. This could further reduce the overall run-time penalty.

Performance of a fault-tolerant code will always be a trade-off between the run-time penalty (extra computational costs) and recovery rate. By adding even more synchronization calls, the recovery rate of the fault-tolerant fluid solver could be increased even further. That in turn, however, would further increase the run-time penalty. It is a matter of optimization based on statistical investigation, what is the likelihood of each failure scenario and how much run-time penalty does it add to make the code resilient to this failure.

# Chapter 3: Minimization of Memory Access

## 3.1 Introduction

When advancing in time either with explicit or implicit timestepping schemes that are being solved iteratively, the resulting formation of a new residual or right hand side $\mathbf{r}$ always follows a pattern of the form:

$$\mathbf{r}^i = \sum C^{ij} \mathbf{f}_{ij} \tag{3.1}$$

where $\mathbf{f}_{ij}$ denotes the flux between entities $i, j$ and $C^{ij}$ the geometric factors that connect the entities $i, j$. Term 'entity' is used to keep the notation general. Examples of entities could be points, cells or elements. The geometric factors $C^{ij}$ could be associated with faces, edges, or the entries in a matrix. Depending on the spatial discretization and the underlying conservation laws, the fluxes $\mathbf{f}$ and the geometric factors $C^{ij}$ may depend nonlinearly on the vector of unknowns $\mathbf{u}_i$.

One can see from equation 4.1 that the formation of a new residual or right hand side implies at least one complete traversal of the database of points, cells, elements, edges or faces, reflecting at least one pass over the complete mesh per time step. Considerable research has been devoted to reducing the number operations required to form accurate fluxes $\mathbf{f}_{ij}$ - e.g. via approximate Riemann solvers and limiters. This implies that the computational intensity, given by the ratio of floating point operations per memory access, is rather low. On the other hand, the speed of CPUs has advanced much faster than the speed of memory access to RAM. This has led to a crisis in CFD: at present, field solvers are limited by the access speed to RAM. Given the number of accesses to memory per time step, the speed of a field solver can be estimated quite accurately. This observation has been documented

repeatedly [7, 8], and can also be observed in the comparison of speeds achieved between CPUs and GPUs [7, 63]. And given that RAM access speeds are not increasing as rapidly as CPU speeds, most CFD codes 'saturate' the memory bandwidth of modern chips after only a few cores, thus minimizing any benefits from going to a higher number of available cores. It thus appears that the aim that was pursued for several decades: obtain the highest accuracy while minimizing floating point operations may therefore have been replaced in the future by the new aim: **obtain the highest accuracy while minimizing memory access**.

## 3.2 CPU speed and memory access speed advancements over time

CPU speed improvements have been following Moore's law for the past 5 decades. In 1965 Gordon Moore predicted that the transistor count on integrated circuits will double every 2 years [15]. This prediction has been kept alive thanks to many technological advancements. Following Moore's law the CPU performance has been increasing just as steady Figure 3.1. Meanwhile the performance of memory access has been increasing at a much lower rate. This situation has led to a performance gap between the processor and memory Figure 3.2. As the performance between the processor and memory grows, the limiting factor for the overall peak performance changes. More and more applications are becoming memory bound - the limiting factor of application performance is the memory access speed. Figure 3.3 shows how the performance gap for different types of memory access has changed over the time. This indicates that the problem persists at different scales. Creating memory aware algorithms has the potential of significantly increasing the overall performance of an application.

## 3.3 Minimizing memory access

Let us consider ways of minimizing memory access for simple finite difference solvers. Starting with the right-hand-side (RHS) for a Laplacian in 1-D, assuming a uniform mesh size.

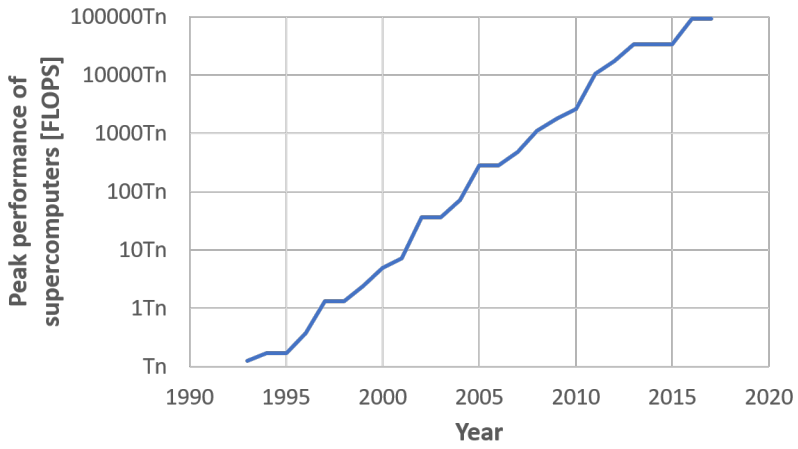



Figure 3.1: Increase in peak performance of supercomputers [16].

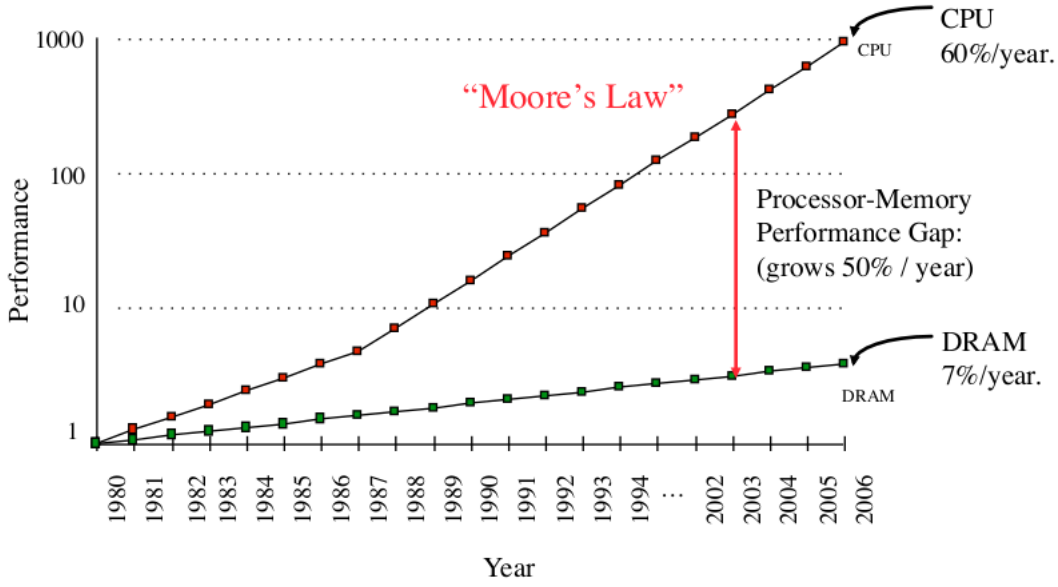



Figure 3.2: Increasing gap between Memory access and CPU performance [64].

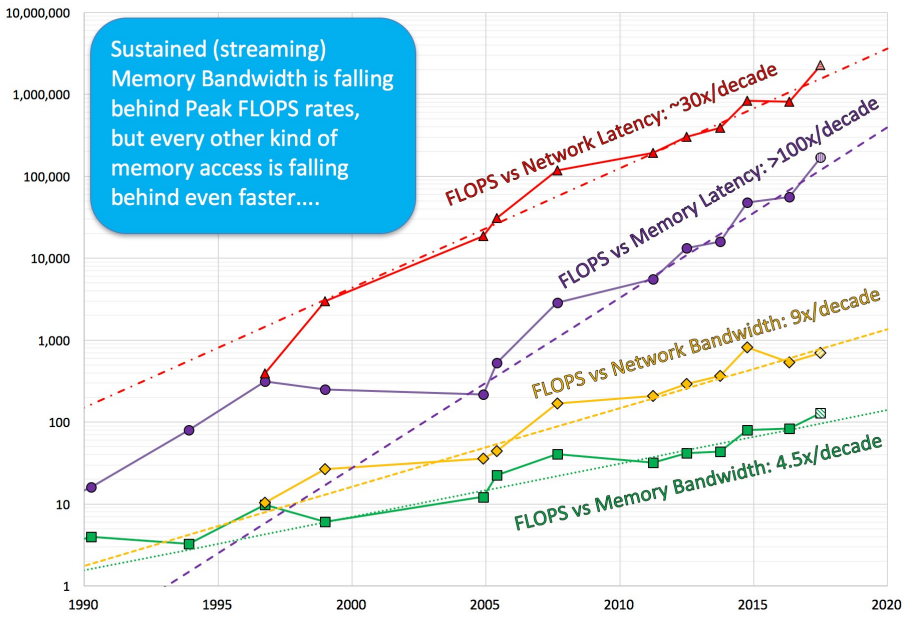



Figure 3.3: Increase in the gap between Memory access and peak performance [65].

The standard 2nd order discretization yields:

$$rhs_i = \frac{1}{\Delta x^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) \tag{3.2}$$

If coded in the usual way as:

```
Loop 1:
        do ipoin=ipoi0,ipoi1
           rhspo(ipoin)=const*(unkno(ipoin-1)-2.0*unkno(ipoin) +unkno(ipoin+1))
        enddo
```

this requires, for each $i$, 3 fetches and 1 store, i.e. 4 accesses to memory. Alternatively, this may be coded as:

```
Loop 2:
       unkn0=unkno(ipoi0-1)
       unkp1=unkno(ipoi0 )
       do ipoin=ipoi0,ipoi1
          unkm1=unkn0
          unkn0=unkp1
          unkp1=unkno(ipoin+1)
          rhspo(ipoin)=const*(unkm1-2.0*unkn0+unkp1)
       enddo
```

While not vectorizable, this requires, for each $i$, 1 fetch and 1 store, i.e. only 2 accesses to memory. The implicit assumption made here and in the following is that the temporary values of `unkm1, unkn0, unkp1` are stored in registers or cache, and thus do not have to be retrieved from memory. In the sequel, loops written in this way will be denoted as minimal memory access loops (MMALs).

The difference in the number of items fetched becomes more pronounced as the stencil (and thus the accuracy of the spatial discretization) increases. The standard 4th order discretization yields:

$$rhs_i = \frac{1}{\Delta x^2}\left(-u_{i-2} + 16u_{i-1} - 30u_i + 16u_{i+1} - u_{i+2}\right) \tag{3.3}$$

If coded as:

```
Loop 3:
       do ipoin=ipoi0,ipoi1
          rhspo(ipoin)=const*(     -unkno(ipoin-2)+ 16.0*unkno(ipoin-1)
     &                        -30.0*unkno(ipoin  )+ 16.0*unkno(ipoin+1)
     &                             -unkno(ipoin+2))
       enddo
```

this requires, for each $i$, 5 fetches and 1 store, i.e. 6 accesses to memory.

Alternatively, this may be coded as a MMAL:

```
Loop 4:
        unkm1=unkno(ipoi0-2)
        unkn0=unkno(ipoi0-1)
        unkp1=unkno(ipoi0 )
        unkp2=unkno(ipoi0+1)
        do ipoin=ipoi0,ipoi1
           unkm2=unkm1
           unkm1=unkn0
           unkn0=unkp1
           unkp1=unkp2
           unkp2=unkno(ipoin+2)
           rhspo(ipoin)=const*(unkm2+16.0*unkm1-30.0*unkn0 +16.0*unkp1-unkp2)
        enddo
```

While not vectorizable, this requires, as before, for each $i$, 1 fetch and 1 store, i.e. only 2 accesses to memory. Remarkably, coding in this way allows to form right-hand sides whose memory access is independent of the approximation order. For every fetch there is one store.

## 3.4 Extension to 2/3D

The situation outlined above is not as favorable in 2/3-D, as the data layout is only optimal in one of the dimensions (usually the first). For the 2nd order stencil

$$rhs_i = \frac{1}{\Delta x^2}\left(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} - 6u_{i,j,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1}\right) \qquad (3.4)$$

the traditional form results in:

```
Loop 5:

       do ipoin=ipoi0,ipoi1

          rhspo(ipoin)=const*(    unkno(ipoin-npoxy)+unkno(ipoin-npoix)

      &                       +    unkno(ipoin-    1)

      &                       -6.0*unkno(ipoin      )

      &                       +    unkno(ipoin    +1)+unkno(ipoin+npoix)

      &                       +    unkno(ipoin+npoxy))

       enddo
```

requiring, for each $i$, 7 fetches and 1 store. If done (as is usually the case) in 3 1-D loops this increases to

- do x: 3 fetches, 1 store
- do y: 4 fetches, 1 store
- do z: 4 fetches, 1 store

i.e. a total of 11 fetches and 3 stores.
In general, given a stencil of size `1+s` in 1-D, the 3D counts result in `1+3s` fetches and `1` store for the unsplit scheme, and `3*(1+s)+2=5+3s` fetches and `3` stores for the split scheme.

For MMALs, several alternatives are possible.

Alternative 1: Gather/Scatter With Inner MMALs:
The idea here is to reorder the arrays using gather/scatter operands and rewrite 3 MMALs (one for each dimension) as before. This results in:

- reorient in y,z, storing in $u_y, u_z$ : 2 fetch, 2 store
- do x with $u$ : 1 fetch, 1 store
- reorient rhs : 1 fetch, 1 store
- do y with $u_y$ : 2 fetch, 1 store

- reorient rhs : 1 fetch, 1 store

- do z with $u_z$ : 2 fetch, 1 store

- reorient rhs : 1 fetch, 1 store

i.e. a total of 10 fetches and 8 stores.

Alternative 2: Inner MMALs With Indirect Addressing:

The idea here is to keep the arrays of unknowns and rhsides untouched, but form a different rhside for each dimension using a pointer array `ldimn(1:npoin)` to locate the proper point as the MMAL is tranversed. For a second order stencil, it would take the following form:

```
Loop 6:

       do ipoin=ipoi0,ipoi1
          unkm1=unkn0
          unkn0=unkp1
          jpoin=jpoi1
          jpoi1=ldimn(ipoin+1)
          unkp1=unkno(jpoi1)
          rhspo(jpoin)=const*(unkm1-2.0*unkn0+unkp1)
       enddo
```

While not vectorizable, this requires, for each $i$, 1 indirect fetch and 1 indirect store, i.e. only 2 accesses to memory. For the complete set of 3-D loops the counts are as follows:

- do x with usual MMAL : 1 da/fetch, 1 da/store

- do y with i/a MMAL : 1 ia/fetch, 1 ia/store

- do z with i/a MMAL : 1 ia/fetch, 1 ia/store

- add rhside : 3 da/fetch, 1 da/store

i.e. a total of 6 fetches and 4 stores, **regardless of the order of the approximation or the stencil size**. The required memory access figures have been compared in Table 1

for stencils of different order. Note the MMAL2 is competitive even with stencils as low as 2.

Table 3.1: Memory Access Counts for FD Schemes of Different Order

| stencil | 3D Unsplit | 3D Split | MMAL1 | MMAL2 |
|---|---|---|---|---|
| 2 | 7F/1S | 11F/3S | 10F/8S | 6F/4S |
| 4 | 13F/1S | 17F/3S | 10F/8S | 6F/4S |
| 6 | 19F/1S | 23F/3S | 10F/8S | 6F/4S |
| 8 | 25F/1S | 29F/3S | 10F/8S | 6F/4S |

Before going on, code Loop 6 is re-written in so-called flux form:

```
Loop 6:

       do ipoin=ipoi0,ipoi1

          unkm1=unkn0

          unkn0=unkp1

          flux1=flux2

          jpoin=jpoi1

          jpoi1=ldimn(ipoin+1)

          unkp1=unkno(jpoi1)

          flux2=const*(unkp1-unkn0)

          rhspo(jpoin)=flux2-flux1

       enddo
```

## 3.5 MMALS for systems of equations

As seen from Table 1, inner MMALs with indirect addressing as exemplified by `Loop 6`, `Loop 7` offer the lowest access rates to memory per residual formed. However, while it is a simple matter to write a loop such as `Loop 7` for a scalar Laplacian with a second order stencil, systems of equations, such as those given by conservation laws, often have many

variables per point, and may require stencils of higher order. This can lead to very long, 'chunky', and thus error-prone loops with many scalar temporaries that may exceed the number of registers available. Three different ways to address this problem were pursued:

- Small vectors;

- Small matrix with indirect addressing;

- Scalar temporaries.

In addition to the three scalar loop options named above, three vectorizable loops were implemented:

- Vectorized version of small vectors;

- Vectorized version of scalar temporaries;

- Vectorized 6-point flux stencil version.

### 3.5.1 Small Vectors

For systems of equations, code Loop 7 would result in a loop of the following form:

```
Loop 8:
       do ipoin=ipoi0,ipoi1
          unkm1(1:nunkp)=unkn0(1:nunkp)
          unkn0(1:nunkp)=unkp1(1:nunkp)
          flux1(1:neqns)=flux2(1:neqns)
          jpoin=jpoi1
          jpoi1=ldimn(ipoin+1)
          unkp1(1:nunkp)=unkno(1:nunkp,jpoi1)
          flux2(1:neqns)=flux(unkp1,unkn0)
          rhspo(1:neqns,jpoin)=flux2(1:neqns)-flux1(1:neqns)
       enddo
```

Here `nunkp,neqns` denote the number of variables stored for the vector of unknowns and the number of flux variables (i.e. the number of equations being solved). The code section where the flux is computed, denoted here as `flux(unkp1,unkn0)`, is identical to that of the conventional loop (i.e. the original code).

### 3.5.2 Small Matrix With Indirect Addressing

The re-store of variables when stepping from `ipoin` to `ipoin+1` (i.e. `unkm1(1:nunkp)=unkn0(1:nunkp)`, `unkn0(1:nunkp)=unkp1(1:nunkp)` and more of the same for higher order stencils) may be avoided by using a small matrix of unknowns that is filled in a circular fashion of

```
Loop 8:

       do ipoin=ipoi0,ipoi1

          unkm1(1:nunkp)=unkn0(1:nunkp)

          unkn0(1:nunkp)=unkp1(1:nunkp)

          flux1(1:neqns)=flux2(1:neqns)

          jpoin=jpoi1

          jpop0=jpop1

          jpoi1=ldimn(ipoin+1)

          jpop1=1+mod(ipoin+1,2)

          unknl(1:nunkp,jpop1)=unkno(1:nunkp,jpoi1)

          flux2(1:neqns)=flux(unknl(jpop1),unknl(jpol0))

          rhspo(1:neqns,jpoin)=flux2(1:neqns)-flux1(1:neqns)

       enddo
```

### 3.5.3 Scalar Temporaries

In this case, all the scalar temporaries needed in a loop, together with all unknowns and fluxes required, are written out explicitly and transferred from `ipoin` to `ipoin+1`. As stated

before, this results in very lengthy and unreadable code. However, it was tried in order to assess the relative merits of the three approaches outlined above.

## 3.6 Implementation in FDFLO

The different variants of minimal memory access loops were implemented into FDFLO, a finite difference code that solves the weakly compressible Navier-Stokes equations given by:

$$\frac{1}{c^2}p_{,t} + \rho\nabla\cdot\mathbf{v} = 0 \quad , \tag{3.5}$$

$$\rho\mathbf{v}_{,t} + \rho\mathbf{v}\nabla\mathbf{v} + \nabla p = \nabla\mu\nabla\mathbf{v} + s_{\mathbf{v}} \quad , \tag{3.6}$$

$$\rho c_p T_{,t} + \rho c_p \mathbf{v}\nabla T = \nabla\mathbf{K}\nabla T + s_T \quad , \tag{3.7}$$

where $\rho, \mathbf{v}, p, c, T, \mu, c_p, k, s_{\mathbf{v}}, s_T$ denote the density, velocity vector, pressure, speed of sound, temperature, viscosity, conductivity and source terms. The code offers a variety of spatial and temporal discretization options, and employs a conservative formulation for the fluxes [7, 8].

## 3.7 Results

All the timings reported were carried out with a spatial discretization of 6th order, resulting in stencils of size 7 per dimension. The temporal integration was performed using an explicit, five-stage, low-storage Runge-Kutta integrator.

5.1 Flow Past Square Cylinder This case considers the flow past a square cylinder of dimensions $0.45 \leq x \leq 0.55$, $0.45 \leq y \leq 0.55$, $0 \leq z \leq 0.41$ immersed in the hexagonal domain $0 \leq x \leq 2.5$, $0 \leq y \leq 0.41$, $0 \leq z \leq 0.41$. The gridsize was $\delta x = 0.005$, uniform throughout the domain, resulting in a mesh of approximately 4 Mcells. The inflow

conditions were set as follows: $p = 0, u = 2.25, v = w = 0, T = 0, c = 20,\ mu = k = 10^{-3}$. The case was run repeatedly for 1000 time steps in order to obtain reliable timings. A typical solution obtained is shown in figureveloc.

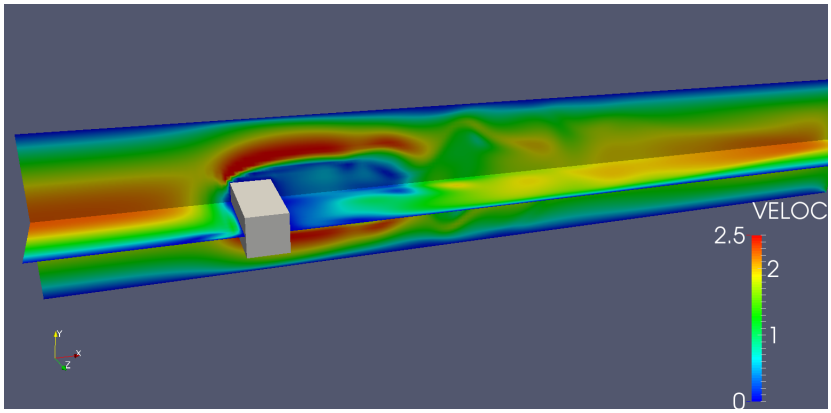



Figure 3.4: Flow Past Square Cylinder

The timings obtained on different machines have been compiled in Table 3.2 to Table 3.6. The labels are as follows: `nprol` denotes the number of cores used in OpenMP mode; Orig the original, dimensionally split, conventional 6th order stencil; ScalTemp the MMAL with scalar temporaries; SmallMat the approach with a small matrix with indirect addressing (Loop 8); SmallVec the small vectors option (Loop 7); MatVect a vectorized version of SmallVec; VScal a vectorized version of scalar temporaries and FLXS6 a vectorized 6-point flux stencil version.

Table 3.2: Timings on SGI Ice-X, Intel Xeon E5-2699v3, 32 C/N, 2.30GHz (thunder) [Sec]

| `nprol` | Orig | ScalTemp | SmallMat | SmallVec | MatVect | VScal | FLXS6 | 2FLXS2 |
|---|---|---|---|---|---|---|---|---|
| 4 | 923 | 1894 | 1403 | 2282 | 1618 | 1321 | 1709 | 3409 |
| 8 | 931 | 2094 | 1474 | 2384 | 1421 | 1022 | 1420 | 3343 |
| 16 | 561 | 1156 | 885 | 1620 | 862 | 619 | 858 | 1816 |
| 32 | 337 | 669 | 543 | 812 | 449 | 411 | 527 | 975 |

Table 3.3: Timings on Laptop, Intel Xeon E31505Mv5, 2.80GHz (rossini) [Sec]

| nprol | Orig | ScalTemp | SmallMat | SmallVec | MatVect | VScal | FLXS6 | 2FLXS2 |
|---|---|---|---|---|---|---|---|---|
| 1 | 1756 | 3616 | 3892 | 4564 | 2524 | 1656 | 2368 | 9296 |
| 4 | 914 | 1252 | 1325 | 1508 | 1112 | 802 | 1048 | 2713 |

Table 3.4: Timings on Desktop, 8 Cores, Intel Xeon E5-2637v4, 3.50GHz (purcell) [Sec]

| nprol | Orig | ScalTemp | SmallMat | SmallVec | MatVect | VScal | FLXS6 | 2FLXS2 |
|---|---|---|---|---|---|---|---|---|
| 1 | 3472 | 7616 | 5704 | 8742 | 7312 | 5912 | 5416 | 6944 |
| 4 | 1062 | 2152 | 2018 | 2725 | 1742 | 1672 | 1698 | 1940 |
| 8 | 944 | 2029 | 1700 | 2549 | 1250 | 1155 | 1280 | 1817 |

Table 3.5: Timings on Cray XC40/50, 544 Intel Xeon Phi 7230 (Knights Landing), 1.30GHz (onyx) [Sec]

| nprol | Orig | ScalTemp | SmallMat | SmallVec | MatVect | VScal | FLXS6 | 2FLXS2 |
|---|---|---|---|---|---|---|---|---|
| 16 | 613 | 1576 | 1591 | 1855 | 939 | 616 | 571 | 4230 |
| 32 | 404 | 1124 | 998 | 1174 | 593 | 406 | 401 | 2792 |
| 64 | 201 | 609 | 604 | 670 | 376 | 252 | 244 | 1479 |

Table 3.6: Timings on SGI ICE-XA, Xeon E5-2698v4, 40 C/N, 2.20GHz (centennial) [Sec]

| nprol | Orig | ScalTemp | SmallMat | SmallVec | MatVect | VScal | FLXS6 | 2FLXS2 |
|---|---|---|---|---|---|---|---|---|
| 4 | 832 | 1888 | 1726 | 2386 | 1556 | 1146 | 1439 | 3380 |
| 8 | 794 | 1870 | 1676 | 2361 | 1183 | 854 | 1091 | 3423 |
| 16 | 438 | 1057 | 967 | 1323 | 674 | 495 | 653 | 1791 |
| 32 | 257 | 581 | 546 | 725 | 382 | 306 | 385 | 941 |

Table 3.7: Timings on Desktop, 32 Cores, AMD Opteron 6238, 2.6GHz (loki) [Sec]

| nprol | Orig | ScalTemp | SmallMat | SmallVec | MatVect | VScal | FLXS6 |
|---|---|---|---|---|---|---|---|
| 1 | 162 | 246 | 182 | 232 | 249 | 175 | 185 |
| 4 | 233 | 378 | 375 | 393 | 344 | 299 | 271 |
| 8 | 397 | 674 | 584 | 645 | 658 | 566 | 442 |
| 16 | 362 | 546 | 519 | 630 | 454 | 445 | 414 |
| 32 | 437 | 471 | 456 | 497 | 430 | 428 | 479 |

Table 3.8: Timings on Desktop, 16 Cores, Intel Xeon Silver 4208, 2.10GHz (tallis) [Sec]

| nprol | Orig | ScalTemp | SmallMat | SmallVec | MatVect | VScal | FLXS6 |
|---|---|---|---|---|---|---|---|
| 1 | 104 | 227 | 250 | 297 | 160 | 155 | 141 |
| 4 | 65 | 161 | 169 | 178 | 87 | 86 | 79 |
| 8 | 292 | 673 | 712 | 814 | 385 | 391 | 352 |
| 16 | 254 | 460 | 453 | 467 | 307 | 282 | 280 |

Table 3.2 - Table 3.6 show preliminary timings carried out in 2018. Table 3.7 and Table 3.8 add timings on more modern computer architectures carried out recently. One can see that none of the minimal memory access loops is faster than the original, dimensionally split, conventional loop. The only exceptions being MMAL with 6-point flux stencil (FLXS6), carried out on 32 Intel Xeon Phi 7230 (Knights Landing) cores and Vectorized small vector MMAL carried out on 32 AMD Opteron 6238 cores. But even in these cases the difference with the conventional loop is insignificant. In general some MMAL options approach the conventional loop for large number of cores, indicating that when memory traffic becomes the bottleneck, the proposed MMAL options may become competitive.

One of the main reasons the MMALs are not outperforming the conventional loop as predicted based on the reduced memory access is the work done by the compiler. The conventional loop can be vectorized thereby increasing the performance. Table 3.9 and Table 3.10 show a more detailed profiling test results. Table 3.9 shows the performance of the main calculation loop of each MMAL subroutine. Table 3.10 shows the overall performance of the MMAL subroutine including the necessary array and vector rearrangements. The conventional loop is vectorized with a 51% efficiency and an estimated speedup gain of 2.06 due to vectorization. Meanwhile the Scalar temporaries, Small vectors and Small matrix MMALs are scalar therefore not possible to vectorize. That can be clearly seen in the Self GFLOPS column showing the performance of each loop in Giga floating point operations per second. The scalar loops have more than 5 times lower operation count per second! The low performance overshadows the possible gain of reduced memory access. Meanwhile, the vectorized small vector main loop operates with the highest operation count 24.82 Gflops.

However, the overall operation count for the whole subroutine is only 6.5 Gflops, due to the necessary data rearrangements.

Table 3.9: Vectorization and performance of the main loops of MMAL subroutines

| Loop | Self time [s] | Efficiency | Gain Estimate | Vector length | Self GFLOPS |
|---|---|---|---|---|---|
| Orig | 15.9 | 51 | 2.06 | 4 | 16.31 |
| ScalTemp | 43.7 | scalar loop | | | 3.26 |
| SmallMat | 47.3 | scalar loop | | | 3.01 |
| SmallVec | 56.8 | scalar loop | | | 2.50 |
| MatVect | 5.4 | 47 | 1.86 | 4 | 24.82 |
| VScal | 8.1 | 41 | 1.65 | 4 | 20.89 |
| FLXS6 | 5.8 | 57 | 2.27 | 4 | 23.07 |

Table 3.10: Vectorization and performance of the MMAL subroutines

| Loop | Self time [s] | Self GFLOPS |
|---|---|---|
| Orig | 16.0 | 16.2 |
| ScalTemp | 43.7 | 3.3 |
| SmallMat | 47.3 | 3.0 |
| SmallVec | 56.8 | 2.5 |
| MatVect | 21.0 | 6.5 |
| VScal | 21.4 | 8.0 |
| FLXS6 | 18.7 | 7.4 |

The columns in Table 3.9 and Table 3.10 denote the following quantities:

- Loop – name of the conventional or MMAL;
- Self time – total time spent in the corresponding subroutine or main loop in seconds;
- Efficiency – calculated performance estimated gain compared to maximum achievable gain from vectorization;
- Gain Estimate – calculated estimate of relative loop performance speedup achieved due to vectorization;

- Vector length – the number of elements processed in a single iteration of vector loops, or the number of elements processed in individual vector instructions. The maximum possible vector length depends on the size of the register;
- Self GFLOPS – ratio of Giga floating-point operations to Self time;

## 3.8 Conclusions and outlook

A number of minimal memory access loop (MMAL) options for finite difference solvers have been described and implemented. The best of these (MMAL2) yields one residual for 6 fetches and 4 stores, regardless of the size of the stencil (and therefore the discretization order). This means that in terms of memory access MMAL2 is competitive even with stencils as low as 2 (typical of CFD codes with 2nd order spatial discretization of fluxes and 4th order damping). Timings for a low Mach number finite difference code using a 6th order spatial discretization show that even though the conventional loops are faster, some implementations approach the speed of the conventional loops for large number of cores.
In theory the MMALs have a clear advantage over the classical implementation with respect to memory access (Table 3.1). That is however disregarding the speedup gained due to vectorization and single instruction multiple data (SIMD) operations that are not available for scalar loops. After taking a closer look at the performance of each proposed MMAL, one can see that in most cases the advantage gained by the reduced memory access is less significant than the performance gain due to vectorization.
Vectorized MMAL options show similar or even higher vectorization efficiency, leading to even higher floating point operation count per second. Here however time is lost performing the necessary data rearrangements to obtain optimal loop execution.
The MMALs perform better on a higher number of cores when the memory bandwidth becomes more 'saturated'. Performance also varies with different hardware architectures. The following chapter presents a method oriented towards targeting the exact underlying CPU architecture.

# Chapter 4: Optimization using Intrinsics

In the previous chapter an optimization approach to minimize memory access has been discussed. One of the results was that the reduced memory access is overshadowed by the more efficient vectorization of the classical loop implementation. However, the vectorization level even in these loops is far from the maximally achievable. A new approach was investigated to see if the full potential of CPU could be utilized. Register intrinsics can be seen as one of the lowest level optimization approaches. It is a tedious and error prone work but if done correctly can yield good results. The aim of this attempt is to maximize the utilization of the registers and increase the peak performance of the subroutine that obtains the RHS for the finite difference approximation of the weakly incompressible Navier-Stokes equations.

## 4.1 Intrinsics

### 4.1.1 SIMD

SIMD stands for Single Instruction Multiple Data. It originated along with the vector supercomputers in the early 70s. Now SIMD is part of almost any microprocessor. SIMD exploits the data parallelism by performing the same operation on a set of data simultaneously as opposed to several times on single pairs of numerical values. The theoretical performance gain when using SIMD depends on the size of the registers (vector size) which directly correlates with the numbers of values that can be processed in parallel. The vector length has been growing steadily with each generation of the SIMD and is currently at 512 bits. With a vector length of 512 bits it is possible to simultaneously execute the same instruction on 8 double precision or 16 single precision numbers. Additionally to the increasing of the length of the vector more SIMD instructions have been added as well. Also

more advanced instructions making it possible to perform almost any operations solely by using SIMD [66].

To take the advantage and use the SIMD features the code has to be vectorized. Vectorization has to fit the underlying hardware. There are several ways of vectorizing a code. On one hand, there is auto-vectorization implemented in modern compilers, which can be seen as the most effortless. On the other hand there is the option to use intrinsic functions, an option that demands the most effort.

### 4.1.2 Auto-vectorization

Auto-vectorization has several advantages when it comes to gaining performance by using SIMD instructions. Developers do not have to worry about portability since the compiler automatically adapts the code to the underlying hardware architecture during the vectorization process. Meaning, it identifies the available vector length and the code is vectorized using the appropriate generation of SIMD instructions. Developers can choose how much effort is spent on improving vectorization by modifying the code where possible or necessary by manual loop unrolling or memory alignment adjustment to provide more favorable memory access for the vectorized execution.

### 4.1.3 Intrinsic functions

Intrinsic functions are functions that operate directly on the registers. These are low level programming instructions. Using intrinsic functions demand a lot of effort, they are platform and hardware specific and can have portability issues. Nonetheless, intrinsic programming is considered the state-of-the-art approach for achieving maximum performance [66]. In most cases auto-vectorization done by the compiler can not achieve the full theoretical SIMD gain. By using intrinsic functions the developer can explicitly ensure the usage of maximum available vector length for SIMD instructions.

Using intrinsics is a complex task. Different sets of intrinsic functions have to be used for different generations of SIMD extensions. If a new extension comes out, the code has to be

re-written (repeat the difficult process) to take advantage of the newly improved technology [67]. Furthermore, SIMD intrinsic instructions are C style functions and are not available when coding in Fortran.

Before switching to intrinsic programming one has to decide if the possible gain outweighs the required work and possible complications.

## 4.2 Implementation

As mentioned before, using intrinsics can ensure the usage of the maximum available vector length. The motivation for applying intrinsic programming to finite difference stencil calculation is the ability to obtain the RHS value of 8 or 16 points (depending on the precision) simultaneously while accessing each of the necessary memory entries only once.

Due to the total number of unknowns and coefficients necessary for the 4th order stencil calculation, only AVX-512 (Advanced vector extension for SIMD instructions) is considered. CPU's supporting AVX-512 have 32 512 bit registers. Previous generations of SIMD instructions had only up to 16 registers. The higher number of registers is necessary to ensure that each value is only laded once from the memory.

In order to better illustrate the possible gains of intrinsic functions as opposed to the usual way, let us look at the simplified RHS calculation. In the previous chapter examples were given for the number of necessary memory accesses (fetches and stores) that are needed to update a 4th order RHS value using a 5 point stencil. For 1D case 5 fetches and 1 store is necessary if coded the usual way. For 3D case 17 fetches and 3 stores are necessary. If using intrinsic operations one would still need the same amount of fetches and stores however that would simultaneously update the value of RHS at 8 different points if using double precision real numbers. If using single precision real numbers, RHS value at 16 points simultaneously can be updated. To put this in other words, the access to the memory can be reduced 8 or 16 times! These are numbers that do not account for auto-vectorization of traditional loops and necessity to rearrange the arrays of unknowns. However the possibility of real gain is there.

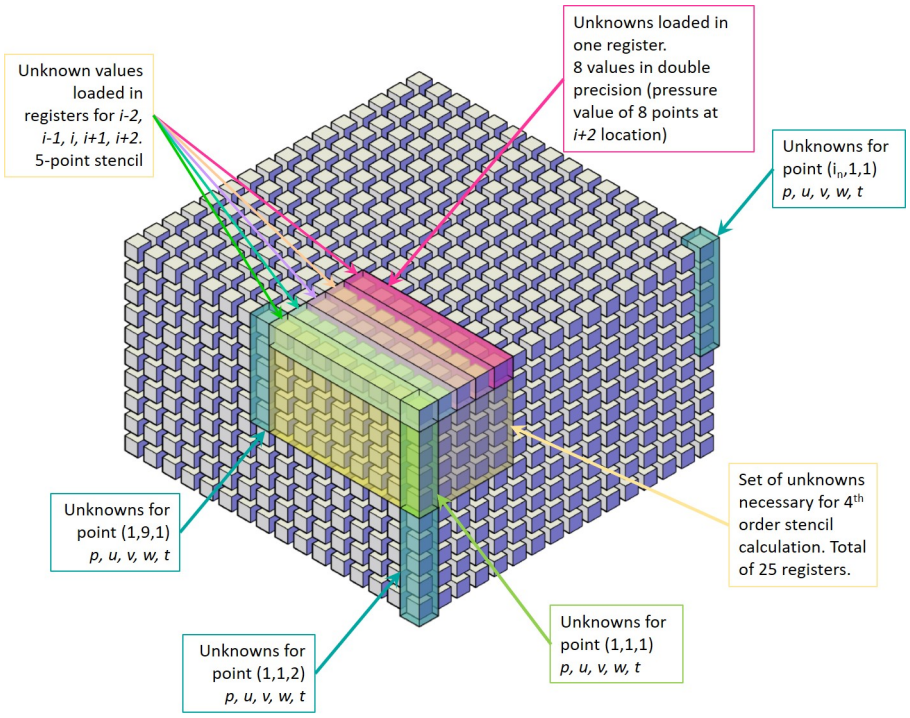



Figure 4.1: Intrinsic memory access to load unknown values in registers (double precision)

Figure 4.1 gives a visual overview of memory access when using AVX-512 SIMD instructions for double precision numbers. It is a schematic representation of the array of unknowns for a domain with 16 points in $x$ and $y$ directions. Halo points are not shown. Access to the array of RHS values and array of conductivity and viscosity values is done in similar way. Figure 4.2 shows the same domain only considering single precision AVX-512 SIMD instructions. Comparison of the two figures shows clear advantage of single precision performance, only one sweep in $x$ direction is necessary to update all the points.

The steps to implement 4th order stencil calculation are described in the following subsections.

### 4.2.1 C++ subroutine

Intel intrinsic instructions for AVX-512 SIMD are C type functions that can not be used in Fortran. A C++ subroutine was written to implement intrinsic RHS value update in the

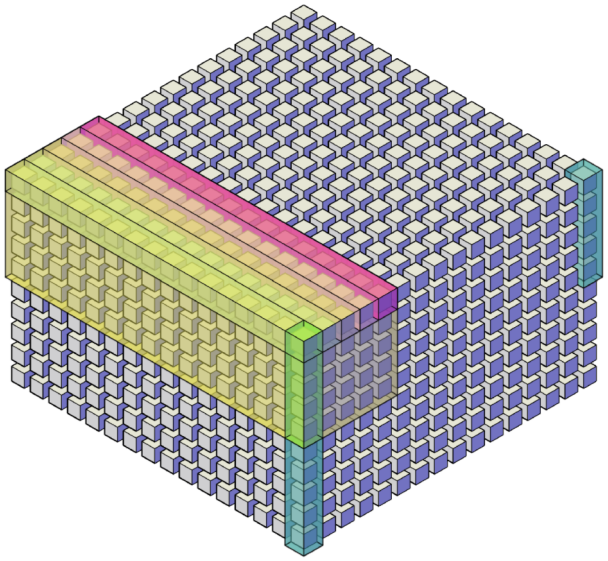


Figure 4.2: Intrinsic memory access to load unknown values in registers (single precision)

finite difference solver. To maximize the efficiency of SIMD instructions all the arithmetic operations have to be executed using intrinsics calls. Regular calls used in between intrinsic instructions can lead to the loss of previously loaded or calculated values in the registers. The overall structure of the C++ subroutine is as follows:

- Obtaining values from the Fortran code;
- Pre-calculating values of coefficients necessary for the RHS calculation;
- Main loop updating all the RHS values, each point is access only once;
- Returning to Fortran an updated RHS value array.

### 4.2.2 Array rearrangement

Possibly one of the main disadvantages of programming with intrinsic instructions is the necessity of having the data arranged in a certain way. To achieve the theoretically maximum performance, each register load from the memory has to populate the register only with usable values. Which means, loading 8 double precision values in a 512 bit register

with one load operation. In order to achieve this certain things have to be fulfilled:

- data arrays have to be aligned so that the 512 bit chunk of data loaded into the register matches exactly to the 512 bit memory space occupied by the corresponding 8 double precision numbers; and,
- the 8 or 16 values needed for the vectorization in the register, follow each other exactly without any other data occupying the memory space in between. When executing the load instruction, one can not choose to load every second or third double precision number in the row. One only specifies the address to the beginning of the 512 bit data chunk that will be loaded in the register.

In order to achieve the necessary data alignment and arrangement all the arrays needed for the RHS update had to be rearranged from the initial Fortran arrangement. The most common and simple way to send an array from Fortran to C++ is as a 1D array (vector). Due to the properties of intrinsic instructions a different data arrangement is necessary for each of the spatial directions. In other words, to update the RHS value using a 3D stencil, three different arrangements of the necessary data is needed.

To minimize the cost of array rearrangement one multilevel-loop was used. Each value in a Fortran array is accessed exactly once and placed in the proper location in the three different direction arrays that are sent to the C++ subroutine.

Another stage of rearrangement is necessary to transfer the data into the original order once the updated RHS arrays (one for each direction) are obtained from the C++ subroutine. A similar multistage loop is used for this purpose.

The cost of the array rearrangement process is discussed in the result section.

## 4.3 Results

In this section the performed tests are discussed. First a smaller test case was investigated to determine the optimal domain size. Later the intrinsic RHS update subroutine was implemented in the FEFLO finite difference code and tested on the TGV case.

### 4.3.1 Test case

A small test case was designed to test the performance of the C++ subroutine using intrinsic instructions. The subroutine obtains the RHS for the finite difference approximation of the weakly incompressible Navier-Stokes equations. The performance of the C++ subroutine was compared with the standard subroutine implemented in Fortran. Different size of domains was tested to see the most favorable configuration. Due to the properties of intrinsic functions all the test domains were of regular cube shape. The number of points in each direction was a multiplication of 8 in case of double precision calculation and a multiplication of 16 for the single precision calculation. In this way the total length of registers can be fully utilized. The results are presented in Table 4.1 and Table 4.2 for double precision and single precision accordingly.

The test case is set up in the following way:

- filling the arrays of the unknowns of the domain points with random numbers;
- running the case for a set number of time steps;
- each time step consists of 4 stages;
- in the case of the C++ subroutine each stage consists of the following operations:
  - rearrangement of the arrays of the unknowns in 3 different arrays that are passed on to the C++ subroutine;
  - obtaining three arrays of the RHS values for the finite difference approximation in C++ subroutine (one for each direction);
  - summation and rearrangement of the arrays of the RHS values to fit the Fortran array arrangement.

The columns in Table 4.1 and Table 4.2 denote the following quantities:

- Subroutine:

Table 4.1: Intrinsic instruction finite difference approximation comparison with standard Fortran subroutine (double precision numbers)

| Sub-routine | Problem size | Self time [s] | Total time [s] | Self GFLOPS | Memory [GB] | ntime | Time per point |
|---|---|---|---|---|---|---|---|
| Fort | 8 | 0.29 | 2.88 | 12.3 | 23 | 5000 | 1.1E-06 |
| C++ | 8 | 0.15 | 0.57 | 21.4 | 9 | 5000 | 2.2E-07 |
| Fort | 16 | 0.44 | 1.03 | 13.0 | 37 | 1000 | 2.5E-07 |
| C++ | 16 | 0.35 | 1.13 | 14.4 | 15 | 1000 | 2.8E-07 |
| Fort | 24 | 1.73 | 2.22 | 11.0 | 125 | 1000 | 1.6E-07 |
| C++ | 24 | 1.19 | 3.67 | 14.3 | 51 | 1000 | 2.7E-07 |
| Fort | 32 | 2.62 | 2.78 | 10.4 | 188 | 500 | 1.4E-07 |
| C++ | 32 | 1.83 | 4.38 | 11.1 | 58 | 500 | 2.8E-07 |
| Fort | 40 | 4.93 | 5.03 | 10.8 | 369 | 500 | 1.6E-07 |
| C++ | 40 | 3.20 | 9.69 | 12.4 | 113 | 500 | 3.0E-07 |
| Fort | 48 | 1.32 | 1.43 | 11.5 | 100 | 100 | 1.3E-07 |
| C++ | 48 | 1.07 | 6.67 | 12.8 | 41 | 100 | 6.0E-07 |

Table 4.2: Intrinsic instruction finite difference approximation comparison with standard Fortran subroutine (single precision numbers)

| Sub-routine | Problem size | Self time [s] | Total time [s] | Self GFLOPS | Memory [GB] | ntime | Time per point |
|---|---|---|---|---|---|---|---|
| Fort | 16 | 1.54 | 3.63 | 20.7 | 81 | 5000 | 1.8E-07 |
| C++ | 16 | 0.73 | 2.34 | 34.8 | 37 | 5000 | 1.1E-07 |
| Fort | 32 | 2.46 | 2.90 | 20.8 | 130 | 1000 | 8.9E-08 |
| C++ | 32 | 1.50 | 4.27 | 26.9 | 60 | 1000 | 1.3E-07 |
| Fort | 48 | 8.60 | 9.58 | 20.1 | 630 | 1000 | 8.7E-08 |
| C++ | 48 | 2.36 | 17.10 | 57.8 | 195 | 1000 | 1.5E-07 |

  - Fort – standard subroutine that obtains the RHS for the finite difference approximation in Fortran;
  - C++ – subroutine that obtains the RHS for the finite difference approximation using intrinsic instructions in C++;

- Problem size – number of points in each direction of a cubic domain;
- Self time – total time spent in the corresponding subroutine in seconds;
- Total time – total runtime of the application in seconds;
- Self GFLOPS – ratio of Giga floating-point operations to Self time;
- Memory – Data transfers between CPU and memory subsystem (total traffic, including caches and DRAM) in gigabytes;
- ntime – number of time steps, each time step includes 4 stages;
- Time per point – time necessary to obtain the RHS of a single point per time step.

It can be seen in the tables that the execution time of the subroutine itself is always shorter in the case of the C++ implementation. As much as two times shorter. Another relation that is constant throughout the whole tests is the total memory transfer. When using intrinsic instructions the amount of memory that needs to be transferred to the CPU is significantly reduced. It is two to three times less than in the case of the standard subroutine in Fortran. One disadvantage of the C++ implementation that clearly presents itself is that the time spent on the arrangement of arrays grows with the problem size. The increase in the overhead due to rearrangement is so high that it overshadows the gain of the more efficient RHS calculation.

The 16x16x16 point domain is chosen for further investigation and implementation in the FDFLO finite difference code. Single precision test of this problem size demonstrates the highest performance (34.8 gflops). Additionally, the total execution time is more than

50 percent shorter even considering the array rearrangement. Problem size of 16 is also reasonable for practical applications.

Although the table shows an even higher performance (57.8 gflops) for the problem size of 48, this result was inconclusive. The profiling tool used showed highly varying results. Furthermore the total execution time with this problem size was almost two times longer for the C++ implementation versus the Fortran.

### 4.3.2 TGV with FDFLO

The subroutine that obtains the RHS for the finite difference approximation using intrinsic instructions in C++ was implemented in FDFLO finite difference code. The performance of this subroutine was tested on a Tailor Green Vortex simulation. The problem set-up and description is given in section 3.11.1. The difference in this test is the domain decomposition. The domain is decomposed in 64 cube form sub-domains with $16 + 2 * 2 = 20$ points in each direction ($2 * 2$ accounts for the two halo points on each side of the domain necessary for the 4th order approximation scheme). Three different versions of the application were executed and compared:

- double precision calculation using Fortran subroutine for obtaining RHS;
- single precision calculation using Fortran subroutine for obtaining RHS;
- single precision calculation using C++ subroutine for obtaining RHS;

Qualitative comparison of the results is shown in Figure 4.3. The single and double precision cases yield very similar results. There is also no significant difference between the single precision calculation using the intrinsic instructions and the single precision calculation using the standard Fortran subroutine.

The timings and performance comparison of the simulations are shown in Table 4.3. One can see that the C++ subroutine has a significant speedup compared to the Fortran subroutine, the Self time is 50% shorter. The floating point operation count per second in C++ subroutine is higher, and the much lower data transfers between CPU and memory

subsystem ensures an overall reduction in execution time. One can also see that the total execution time is still favorable to the standard Fortran implementation. This is explained by the time spent rearranging the arrays of unknowns to fit the intrinsic instruction calculation. The total time spent on data rearrangement is $29s$ which is more than three times higher than the subroutine execution time itself.

Table 4.3: Performance comparison between intrinsic instruction finite difference approximation and standard Fortran subroutine (single precision) for the TGV case

| Loop | Self time [s] | Self GFLOPS | Memory [GB] | Total execution time [s] |
|---|---|---|---|---|
| Fortran | 10.7 | 31.9 | 504.7 | 36.3 |
| C++ | 4.8 | 34.4 | 337.5 | 62.2 |

## 4.4 Conclusions and outlook

Intrinsic instructions are the state of the art approach to achieve maximum performance on a given computational system. It does involve, however, complex low level programming and the final code has limited portability. The presented test case clearly shows the possible gains of RHS update calculation using intrinsic instructions. It results in high floating point operations as a result of maximizing the used vector length. It also reduces the total data transfer from memory to CPU by more than a factor of 2.

The performance increase of a fully intrinsic subroutine comes with additional computational costs spent on the rearrangement of data. As shown in the Tailor Green vortex case, time spent on data rearrangement can even exceed the time spent in the intrinsic subroutine.

To further explore the possible gains of using intrinsic instructions it is necessary to address the following questions and explore further implementation options.

The proposed further research steps are as follows:

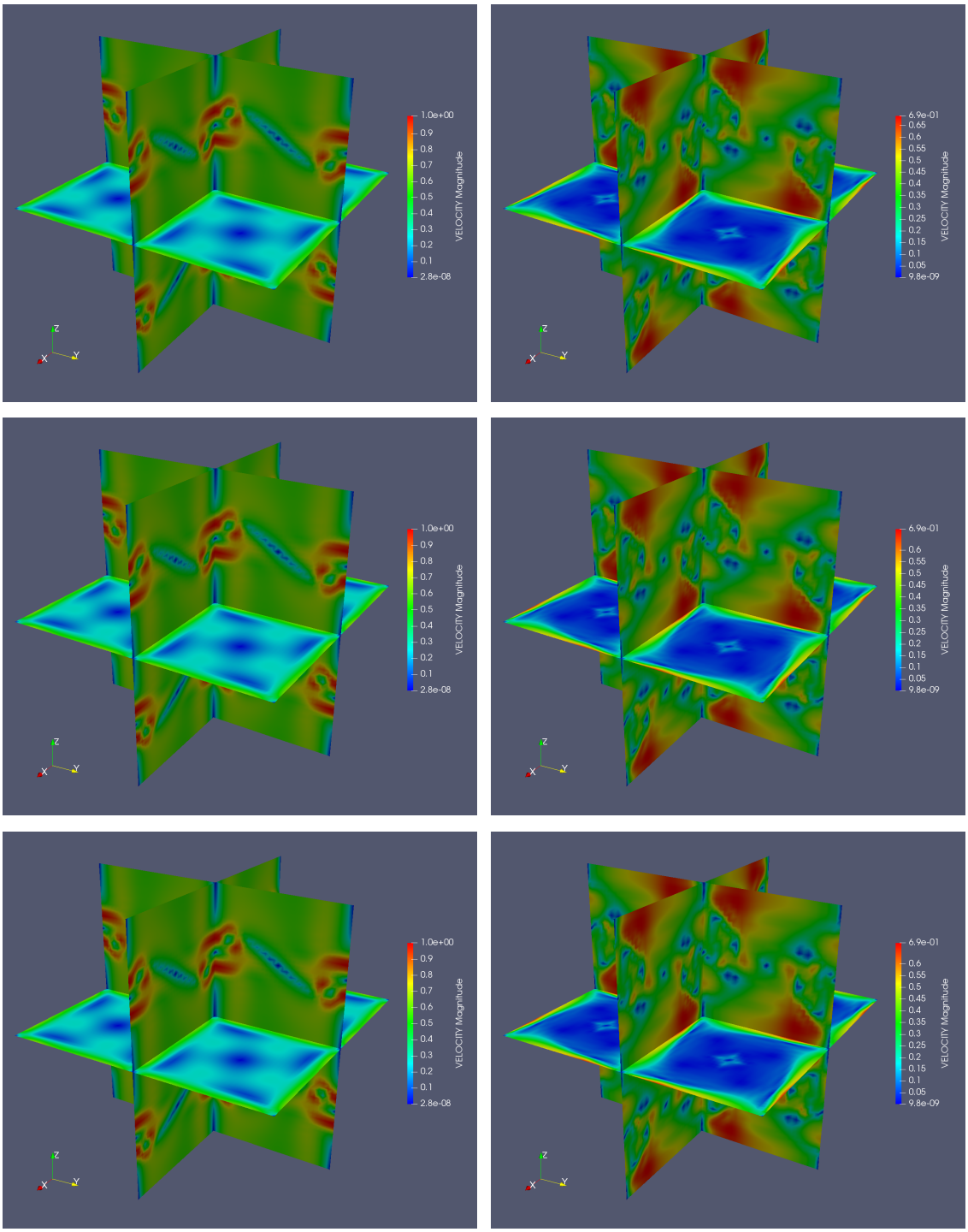



Figure 4.3: TGV: Velocity field at $t = 5s$ (left column) and $t = 10s$ (right column). Comparison of single and double precision results using different subroutines: Double precision with Fortran subroutine (top), single precision with C++ subroutine (middle) and single precision with Fortran subroutine (bottom)

- Optimization of data rearrangement subroutines. Explore more efficient rearrangement algorithms or ways how to reuse the already rearranged data. Minimize the amount of the necessary rearrangements.

- Using the strategies of MMALs when implementing intrinsic instructions. The current implementation reloads the registers with necessary unknowns at each iteration step. Although it already has an advantage over the conventional method, using MMAL method could even further increase the advantage. For the 4th order scheme, instead of having to load 22 sets of unknowns from the memory at each iteration step, one could load only 7 (replace the unknowns of the last point of stencil with the unknowns of the next point). As there is a limited amount of registers per CPU, this is not so straight forward. Having more than 20 registers filled with reusable unknowns at all times does not leave enough free registers for the necessary operations and constants. There are two options that could be explored, however:

  - Reset the register values to the necessary constants multiple times during the iteration. Although not optimal, the set operation only has to access one constant value per call instead of the whole vector length worth of values. The amount of constants necessary is small enough to let this information come from the higher levels of memory - faster access.
  - Less realistic and straight forward option is to expect development of the CPUs with more than 32 registers.

- The rearrangement of arrays takes the major part of the execution time. It does not however grow with the increase in the order of the scheme. Same rearrangement can be used also for the 6th and 8th order schemes. Since the execution time of higher order schemes is higher, intrinsic instruction subroutine could become more competitive.

- Finally one could explore using intrinsic instructions for larger portions of the code, where same data arrangement is necessary. This way the rearranged data can be reused multiple times while increasing the performance of other portions of the code.

Even though the current results show no real advantage of using intrinsic instructions over traditional program calls, the preliminary results show evidence of possible performance increase. If the issues stated above are addressed and solved, intrinsic instructions could lead to a more efficient fluid solver development.

# Chapter 5: Conclusions

The topic of this dissertation is the optimization of fluid solvers with respect to fault tolerance and memory latency. The first issue comes from the challenges associated with exascale computing. As the number of cores and nodes in a supercomputer grows, the mean time between hardware failures reduces. Many of the scientific codes used today are not able to overcome such failures. Each encountered failure leads to an aborted simulation. Then, the simulation needs to be restarted from the last checkpoint written to the disk. Depending on the time of the occurrence, this results in the loss of several hours of productive work. Furthermore, restarting a failed simulation involves the manual work of an engineer.

The first issue addressed in this work concentrates on the development of a fault tolerant fluid solver. The aim is to develop strategies that can be generally applied to fluid solvers in order to achieve fault tolerance. This work proposes simple implementations (extensions) to existing code that do not demand extensive rewriting of the code. Current tests conducted on local machines as well as on several cluster environments yield promising results:

- the suggested fault tolerant fluid solver implementation demonstrates recovery rates above 90%; and,
- large simulationS running on 16 compute nodes and 256 MPI processes have a runtime penalty as low as 12%.

The second issue addressed in this work is the widening gap between the performance of CPUs and the performance of memory access speeds. Applications like finite difference solvers that access large amounts of data at each iteration become memory bound. The limiting factor of the execution speed is the access speed to the memory. Memory-aware algorithms are needed to reach peak CPU performance.

Several minimal memory access loop options are proposed in this work. Once loaded in higher levels of memory, data is re-used. The current results do not show a clear advantage of the proposed loops since they cannot be vectorized with the same efficiency as the conventional loops. At the same time, the tests prove the existence of the memory latency problem. The MMAL loops become more competitive with conventional methods when running on a higher number of cores–memory bandwidth becomes more 'saturated.' In some test scenarios, the MMAL loops even outperform the conventional loops.

Finally, a subroutine that obtains the RHS for the finite difference approximation using intrinsic instructions in C++ was implemented in FDFLO finite difference code. Intrinsic instructions ensure usage of the maximum vector, meaning the maximum simultaneous operations performed at a given CPU cycle. It also reduces the total memory traffic between the CPU and different levels of memory. The intrinsic instruction subroutine outperforms the conventional method, however, it loses its overall advantage due to necessary data rearrangements. Although the current test results do not yield the expected speedup, they show evidence of possible performance increase that could be achieved with future work.

# Appendix A: Fault-tolerant code extensions

## A.1 Fault repair subroutine

The following subroutine is called by all active processes of the MPI communicator after a process failure is encountered. As basis for this subroutine the suggested call sequence of ULFM is used [68].

```
1
2         subroutine mpp_comm_replace(MPI_COMM0,MPI_COMM_new,lcfdc  ,ierro)
3 c
4         use arrays_mpp
5 c
6         implicit real*8 (a-h,o-z)
7 c
8         include 'mpp.h'
9 c
10 c      ------mpp common
11 c
12        common /mpp_info/ nproc,iproc,ilang,nprol
13        common /mpp_err/ MPI_ERR
14        common /mpp_comm/ MPI_COMM
15 c
16        integer*4 info,isize,irank
17        integer*4 ierro,isv_key,namelen
18        logical   flag
19        character*(MPI_MAX_PROCESSOR_NAME) pname
20 c
21        external mpp_error_handler
22 c
23        integer lcfdc(6)
```

```
c
c     ------this sub fixes the communicator once processors have failed
c
c          on input : MPI_COMM0   : original world communicator
c                     lcfdc: 1: cfd code is active (=1)
c                            2: nr. or (mpi) processors desired: nproc
c                            3: new_group MPI_COMM
c                            4: new_numtasks
c                            5: new_rank (iproc)
c                            6: original/new iproc indicator
c                               0: original
c                               1: new iproc
c
c          on output: MPI_COMM0   : new world communicator
c                     lcfdc: 1: cfd code is active (=1)
c                            2: nr. or (mpi) processors desired: nproc
c                            3: new_group MPI_COMM
c                            4: new_numtasks
c                            5: new_rank (iproc)
c                            6: original/new iproc indicator
c                               0: original
c                               1: new iproc
c
c     ------for fault-tolerant tests
c
        write(6,*)' in mpp_comm_replace: before repartition'
        write(6,10)(lcfdc(i),i=1,6)
   10 format(' lcfdc(1)= ',i12,
     &      /,' lcfdc(2)= ',i12,
     &      /,' lcfdc(3)= ',i12,
     &      /,' lcfdc(4)= ',i12,
```

```
55      &      /,' lcfdc(5)= ',i12,
56      &      /,' lcfdc(6)= ',i12)
57  c
58        nproc_needed=lcfdc(2)
59        MPI_COMM_cfd=lcfdc(3)
60  c
61  c     ------shrink to alive procs (new communicator)
62  c
63        call mpix_comm_shrink(MPI_COMM0,MPI_COMM_new,ierr)
64  c
65  c     ------see how many procs died
66  c
67        call     mpi_comm_size(MPI_COMM_new,nproc_new,iinfo)
68        call     mpi_comm_size(MPI_COMM0    ,nproc    ,iinfo)
69        ndead=nproc-nproc_new
70        write(6,*)' nr. of dead processors: ndead= ',ndead
71  c
72        if(ndead.eq.0) then
73                                                          goto 9999
74        endif
75  c
76  c     ------see if we have sufficient spare processes
77  c          and abort if necessary
78  c
79        if(nproc_new.lt.nproc_needed) then
80           write (6,*)' no more sufficient active procs ==> stopped'
81           write (6,*)' nproc_new,nproc_needed= ',nproc_new,nproc_needed
82           call mpi_abort(MPI_COMM_WORLD,ierro)
83        endif
84  c
85  c     ------set the error handler for the new communicator
```

```
c           to MPI_ERRORS_RETURN
c
     &             ' MPI_COMM_new= ',MPI_COMM_new
        call mpi_comm_set_errhandler(MPI_COMM_new,MPI_ERRORS_RETURN,ierr)
c
c     ------get the proper ranks in new world
c
        call mpi_comm_rank(MPI_COMM0    ,irank0,iinfo)
        iproc0=irank0+1
c
        call mpi_comm_rank(MPI_COMM_new,irank1,iinfo)
        iproc1=irank1+1
c
        write(6,*)' iproc,iproc0,iproc1= ',iproc,iproc0,iproc1
        write(6,*)'        irank0,irank1= ',irank0,irank1
c
        call mpi_barrier(MPI_COMM_new,ierr)
c
        write(6,*)' after    mpp_barrier (1): MPI_COMM_new= ',MPI_COMM_new
c
c     ------the rank 0 in MPI_COMM0,MPI_COMM_new is going to
c          determine the ranks at which the reserve procs
c          need to be inserted
c
        np=lcfdc(2)            ! nr of procs needed for running cfd code
c
        write(6,*)' nproc,np= ',nproc,np
c
        if(irank1.eq.0) then
c
c     ------get the group of dead procs ==>
```

```
117 c           those in MPI_COMM0, but not in MPI_COMM_new are the dead ones
118 c
119           call        mpi_comm_group(MPI_COMM0    ,igrp0, ierr)
120           write(6,*)' igrp0= ',igrp0
121 c
122       &               MPI_COMM_new
123           call        mpi_comm_group(MPI_COMM_new,igrp1, ierr)
124           write(6,*)' igrp1= ',igrp1
125 c
126           call mpi_group_difference(igrp0, igrp1, igrp2, ierr)
127 c
128 c     ------compute the rank assignment for the newly inserted spares
129 c           number of dead active cfd processes
130 c
131           indw=0
132 c
133           do 1200 i=0,ndead-1
134           call mpi_group_translate_ranks(igrp2,1,i,igrp0,idrank,ierr)
135 c
136 c     ----if the dead process was an active cfd process send the
137 c           information to the spare, else ignore
138 c
139           write(6,*)' idrank,np= ',idrank,np
140 c
141              if(idrank.lt.np) then
142                 write(6,*)' in send do loop: ',idrank,nproc_new-(i+1)
143                 call mpi_send(idrank,1,MPI_INT,nproc_new-(i+1),1,
144       &                       MPI_COMM_new,ierr)
145                 indw=indw+1
146              else
147                 write(6,*)' spare died, no action taken'
```

```
148               endif
149               write (6,*) 'after sending the new ranks',indw,
150      &                    'procs replaced'
151   1200    continue
152   c
153   c     ------free the groups
154   c
155             call mpi_group_free(igrp0, ierr)
156             call mpi_group_free(igrp1, ierr)
157             call mpi_group_free(igrp2, ierr)
158   c
159           endif
160   c
161   c     ------broadcast the number of procs needed to be replaced
162   c
163           call mpi_bcast(indw,1,MPI_INT,0,MPI_COMM_new,ierr)
164   c
165           write(6,*)' calling mpp_barrier (2): MPI_COMM_new= ',MPI_COMM_new
166           call mpi_barrier(MPI_COMM_new, ierr)
167   c
168   c     ------loop where the new workers receive the ranks
169   c
170           inewp=0
171   c
172           do 1400 i=0,indw-1
173           if(irank1.eq.nproc_new-(i+1)) then
174              write(6,*)' in receive do loop: ',irank
175              call mpi_recv(irank0,1,MPI_INT,0,1,MPI_COMM_new,
176      &                     MPI_STATUS_IGNORE,ierr)
177              inewp=1                ! new processor
178              write(6,*)'received irank1,irank0: ',irank1,irank0
```

```
179         endif
180         np = np +1
181  1400 continue
182 c
183 c      ------free the old communicator
184 c
185         call mpi_comm_free(MPI_COMM_cfd, ierr)
186 c
187         call mpi_barrier(MPI_COMM_new, ierr)
188 c
189 c      ------just in case...
190 c
191         call  mpi_comm_rank(MPI_COMM_new, irank_new, ierr)
192         call  mpi_comm_size(MPI_COMM_new, nproc     , ierr)
193 c
194 c      ------re-arrange ranks in the MPI_COMM_new
195 c
196         call mpi_comm_split(MPI_COMM_new,0,irank0,MPI_COMM_tmp,ierr)
197         call  mpi_comm_rank(MPI_COMM_tmp,irank_new,ierr)
198        MPI_COMM_new=MPI_COMM_tmp
199         write(6,*)' rank after re-arranging, irank_new ',irank_new
200 c
201 c      ------re-create fixed workcomm using processes from the process pool
202 c
203 c      ------assign a color to the ranks/processes we want
204 c           to insure that we know which ranks failed in the old
205 c           communicator for easier recovery strategy
206 c
207         write(6,*)' calling mpp_comm_split: MPI_COMM_new= ',MPI_COMM_new
208 c
209         if(irank0.ge.0. and .irank0.le.nproc_needed-1) then
```

```
210            icolor=1
211          else
212            icolor=MPI_UNDEFINED
213          endif
214 c
215          call mpi_comm_split(MPI_COMM_new,icolor,irank0,
216         &                    MPI_COMM_cfd,ierr)
217 c
218          write(6,*)' after    mpp_comm_split: MPI_COMM_cfd= ',MPI_COMM_cfd
219 c
220          if(icolor.eq.1) then
221            call  mpi_comm_rank(MPI_COMM_cfd, irank_cfd, ierr)
222            call  mpi_comm_size(MPI_COMM_cfd, nproc_cfd, ierr)
223            iproc=irank_cfd+1
224          else
225            irank_cfd=-1
226            nproc_cfd= 0
227          endif
228          write (6,*) 'after split: irank_cfd,irank_new= ',
229         &                          irank_cfd,irank_new
230          write (6,*) 'after split: nproc_cfd= ',nproc_cfd
231 c
232          if(icolor.eq.1. and .nproc_cfd.ne.lcfdc(2)) then
233            write(6,*)' error: nproc_cfd,lcfdc(2)= ',nproc_cfd,lcfdc(2)
234            write(6,*)' ==> stopped'
235            call mpi_abort(MPI_COMM_WORLD,ierro)
236          endif
237 c
238          lcfdc(3)=MPI_COMM_cfd
239          lcfdc(4)=nproc_cfd
240          lcfdc(5)=irank_cfd+1
```

```
241         lcfdc(6)=inewp
242 c
243         write(6,*)' after repartition'
244         write(6,12)(lcfdc(i),i=1,6)
245    12 format(' lcfdc(1)= ',i12,
246      &       /,' lcfdc(2)= ',i12,
247      &       /,' lcfdc(3)= ',i12,
248      &       /,' lcfdc(4)= ',i12,
249      &       /,' lcfdc(5)= ',i12,
250      &       /,' lcfdc(6)= ',i12)
251 c
252 c      ------restore the error handlers
253 c
254         call mpi_comm_get_errhandler(MPI_COMM0    ,ierrh,ierr)
255         call mpi_comm_set_errhandler(MPI_COMM_new,ierrh,ierr)
256         call mpi_comm_set_errhandler(MPI_COMM_cfd,ierrh,ierr)
257 c
258         write(6,*)' exiting mpp_comm_replace'
259 c
260  9999 continue
261       return
262       end
```

## A.2 Extended fault tolerant MPI all reduce call sequence

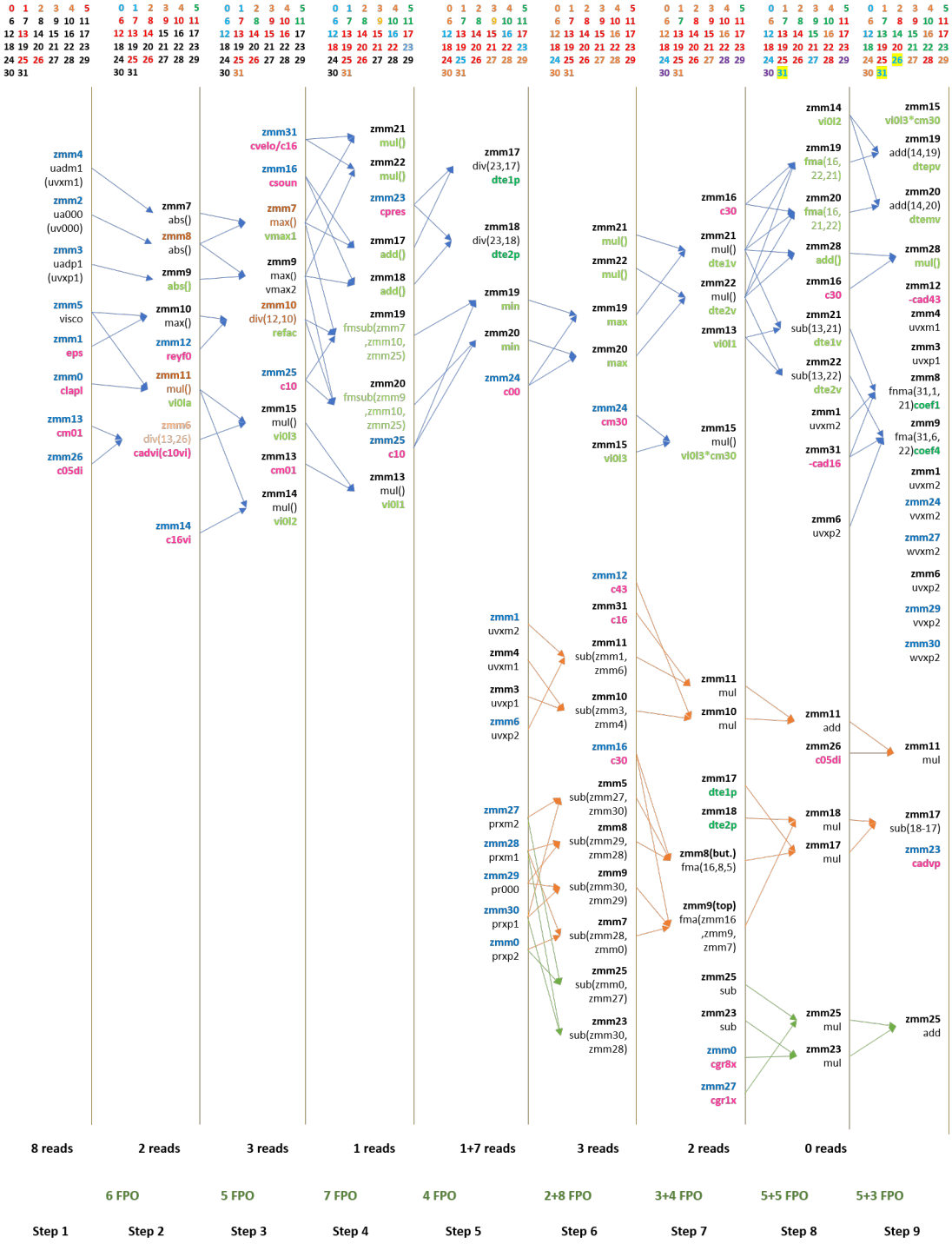
MPI all reduce calls can be prone to stuck processes or processes leaving the call with different status information. While some processes might be notified of a process failure on the communicator, others might see no such error and proceed with the execution. In order to avoid such situation and ensure consensus the following call sequence is implemented.

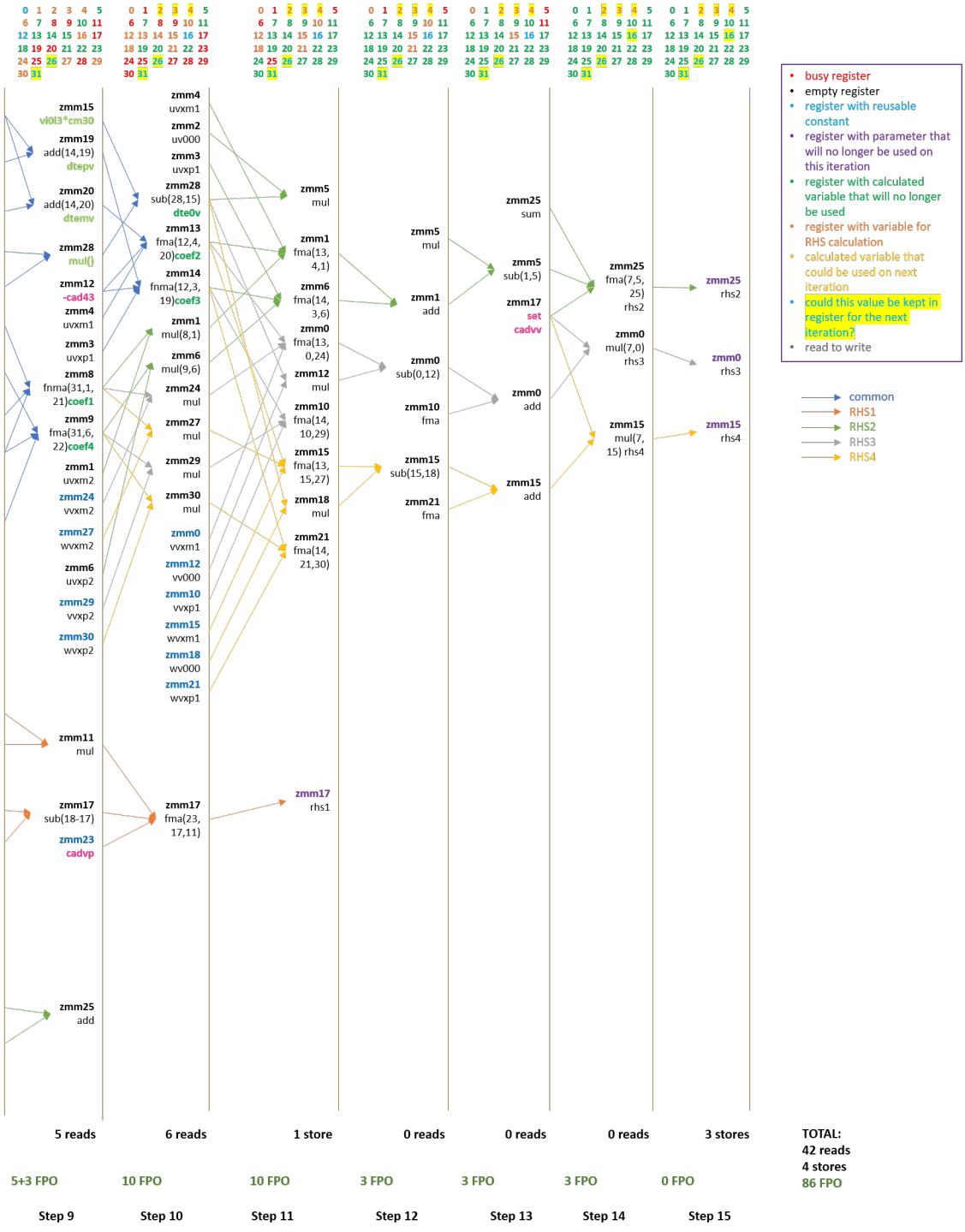
```
1
2 #if defined(WITHFAULTTOLERANCE)
3 c     ---- non-blocking all reduce call is used
4             call mpi_iallreduce(sdata,rdata,ndata,MPI_REAL8,
5      &                          i_mpp_op,MPI_COMM,ireq,ierrmpp)
6 c     ---- mpi barrier to synchronize all processes
7             call mpi_barrier(MPI_COMM,ierrmpp)
8 c     ---- checking if any process has registered an error and getting a consensus'
9             iflag=(MPI_SUCCESS .eq. ierrmpp)
10            call mpix_comm_failure_ack(MPI_COMM, ierrmpp)
11            call mpix_comm_agree(MPI_COMM, iflag, ierrmpp)
12 c     ---- if error has been encountered all the processes return with an error
13           code to the main program
14            if (.not. iflag .or. ierrmpp .ne. 0) then
15               write(6,*)'     setting ierrmpp to 10'
16               ierrmpp=10
17               goto 9999
18            endif
19 c     ---- using mpi_wait to ensure synchronous finalization of the call sequence
20 c          between the processes
21            call mpi_wait(ireq,istat,ierrmpp)
22 #else
23 c     ---- in case of non fault tolerant run, regular blocking mpi all reduce
24 c          call is used
25            call mpi_allreduce(sdata,rdata,ndata,MPI_REAL8,
26      &                        i_mpp_op,MPI_COMM,ierrmpp)
27
28 #endif
```

# Appendix B: Intrinsic instruction call sequence

## B.1 Intrinsic instruction call sequence and register filling

When using intrinsic instructions it is important to keep track of what data is loaded in which register as well as which registers are available to leading new data from the memory without overwriting reusable data. The following flowchart demonstrates the usage of registers and intrinsic calls throughout the RHS update subroutine.


8 reads
2 reads
3 reads
1 reads
1+7 reads
3 reads
2 reads
0 reads
6 FPO
5 FPO
7 FPO
4 FPO
2+8 FPO
3+4 FPO
5+5 FPO
5+3 FPO
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
Step 7
Step 8
Step 9


busy register
empty register
register with reusable constant
register with parameter that will no longer be used on this iteration
register with calculated variable that will no longer be used
register with variable for RHS calculation
calculated variable that could be used on next iteration
could this value be kept in register for the next iteration?
read to write
common
RHS1
RHS2
RHS3
RHS4
zmm15 vi0l3*cm30
zmm19 add(14,19) dtepv
zmm20 add(14,20) dtemv
zmm28 mul()
zmm12 -cad43
zmm4 uvxm1
zmm3 uvxp1
zmm8 fnma(31,1, 21)coef1
zmm9 fma(31,6, 22)coef4
zmm1 uvxm2
zmm24 vvxm2
zmm27 wvxm2
zmm6 uvxp2
zmm29 vvxp2
zmm30 wvxp2
zmm11 mul
zmm17 sub(18-17)
zmm23 cadvp
zmm25 add
zmm4 uvxm1
zmm2 uv000
zmm3 uvxp1
zmm28 sub(28,15) dte0v
zmm13 fma(12,4, 20)coef2
zmm14 fnma(12,3, 19)coef3
zmm1 mul(8,1)
zmm6 mul(9,6)
zmm24 mul
zmm27 mul
zmm29 mul
zmm30 mul
zmm0 vvxm1
zmm12 vv000
zmm10 vvxp1
zmm15 wvxm1
zmm18 wv000
zmm21 wvxp1
zmm17 fma(23, 17,11)
zmm5 mul
zmm1 fma(13, 4,1)
zmm6 fma(14, 3,6)
zmm0 fma(13, 0,24)
zmm12 mul
zmm10 fma(14, 10,29)
zmm15 fma(13, 15,27)
zmm18 mul
zmm21 fma(14, 21,30)
zmm17 rhs1
zmm5 mul
zmm1 add
zmm0 sub(0,12)
zmm10 fma
zmm15 sub(15,18)
zmm21 fma
zmm25 sum
zmm5 sub(1,5)
zmm17 set cadvv
zmm0 add
zmm15 add
zmm25 fma(7,5, 25) rhs2
zmm0 mul(7,0) rhs3
zmm15 mul(7, 15) rhs4
zmm25 rhs2
zmm0 rhs3
zmm15 rhs4
5 reads
6 reads
1 store
0 reads
0 reads
0 reads
3 stores
TOTAL:
42 reads
4 stores
86 FPO
5+3 FPO
10 FPO
10 FPO
3 FPO
3 FPO
3 FPO
0 FPO
Step 9
Step 10
Step 11
Step 12
Step 13
Step 14
Step 15

# Bibliography

# Bibliography


[1] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of roadrunner," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 2008, pp. 1–11.

[2] "Top500 becomes a petaflop club for supercomputers," 2019. [Online]. Available: https://www.top500.org/news/top500-becomes-a-petaflop-club-for-supercomputers/

[3] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[4] N. Romero, E. Jennings, Á. Vázquez-Mayagoitia, V. Vishwanath, and T. Williams, "Alcf data science and machine learning programs: From petascale to exascale," *Bulletin of the American Physical Society*, vol. 63, 2018.

[5] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "Resilience-aware resource management for exascale computing systems," *IEEE Transactions on Sustainable Computing*, vol. 3, no. 4, pp. 332–345, 2018.

[6] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 610–621.

[7] R. Löhner, A. T. Corrigan, K.-R. Wichmann, and W. Wall, "On the achievable speeds of finite difference solvers on cpus and gpus," in *21st AIAA Computational Fluid Dynamics Conference*, 2013, p. 2852.

[8] ——, "Comparison of lattice-boltzmann and finite difference solvers," in *52nd Aerospace Sciences Meeting*, 2014, p. 1439.

[9] R. Löhner, A. Figueroa, and A. Degro, "Recent advances in a cartesian solver for industrial les," in *AIAA Scitech 2019 Forum*, 2019, p. 2328.

[10] R. Löhner, C. Othmer, M. Mrosek, A. Figueroa, and A. Degro, "Overnight industrial les for external aerodynamics," in *AIAA Scitech 2020 Forum*, 2020, p. 2031.

[11] R. Löhner and J. D. Baum, "Handling tens of thousands of cores with industrial/legacy codes: Approaches, implementation and timings," *Computers and Fluids*, vol. 85, 10 2013.

[12] R. Löhner and J. Baum, "On maximum achievable speeds for field solvers," *International Journal of Numerical Methods for Heat & Fluid Flow*, vol. 24, 08 2014.

[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[14] R. Löhner, *Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods*. Wiley, 2008.

[15] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965.

[16] "Top500 list." [Online]. Available: https://www.top500.org/

[17] K. Rupp, "42 years of microprocessor trend data." [Online]. Available: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

[18] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conference Series*, vol. 78, no. 1. IOP Publishing, 2007, p. 012022.

[19] R.-T. Liu and Z.-N. Chen, "A large-scale study of failures on petascale supercomputers," *Journal of computer science and technology*, vol. 33, no. 1, pp. 24–41, 2018.

[20] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[21] F. Petrini, K. Davis, and J. C. Sancho, "System-level fault-tolerance in large-scale parallel machines with buffered coscheduling," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, 2004, p. 209.

[22] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: long-term measurement, analysis, and implications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[23] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

[24] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2009.

[25] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *The international journal of high performance computing applications*, vol. 25, no. 1, pp. 3–60, 2011.

[26] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 25–36.

[27] F. Cappello and G. Bosilca, "Application-driven fault-tolerance for high performance distributed computing." [Online]. Available: https://fault-tolerance.org/wp-content/uploads/2018/08/europar18-Introduction.pdf

[28] "MPI: A message-passing interface standard," p. 852. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[29] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou, "Mpi-ft: Portable fault tolerance scheme for mpi," *Parallel Processing Letters*, vol. 10, no. 04, pp. 371–382, 2000.

[30] G. E. Fagg and J. J. Dongarra, "Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2000, pp. 346–353.

[31] M. Beck, J. J. Dongarra, G. E. Fagg, G. Al Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous *et al.*, "Harness: A next generation distributed virtual machine," *Future Generation Computer Systems*, vol. 15, no. 5-6, pp. 571–582, 1999.

[32] G. E. Fagg, A. Bukovsky, and J. J. Dongarra, "Harness and fault tolerant mpi," *Parallel Computing*, vol. 27, no. 11, pp. 1479–1495, 2001.

[33] "Ft-mpi." [Online]. Available: https://icl.utk.edu/ftmpi/overview/index.html

[34] A. Hassani, A. Skjellum, and R. Brightwell, "Design and evaluation of fa-mpi, a transactional resilience scheme for non-blocking mpi," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 750–755.

[35] A. Hassani, A. Skjellum, P. V. Bangalore, and R. Brightwell, "Practical resilient cases for fa-mpi, a transactional fault-tolerant mpi," in *Proceedings of the 3rd Workshop on Exascale MPI*, 2015, pp. 1–10.

[36] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[37] "Draft document for a standard message-passing interface." [Online]. Available: https://fault-tolerance.org/ulfm/ulfm-specification/

[38] "Mpich overview." [Online]. Available: https://www.mpich.org/about/overview/

[39] "Mpich wiki, fault tolerance." [Online]. Available: https://wiki.mpich.org/mpich/index.php/Fault_Tolerance

[40] "Mpich github issue forum." [Online]. Available: https://github.com/pmodels/mpich/issues/2198

[41] M. Gamble, R. Van Der Wijngaart, K. Teranishi, and M. Parashar, "Specification of fenix mpi fault tolerance library version 1.0." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[42] N. Weeks, G. Luecke, P. Maris, and J. Vary, "Challenges in developing mpi fault-tolerant fortran applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 524–531.

[43] I. Laguna, T. Gamblin, K. Mohror, M. Schulz, H. Pritchard, and N. Davis, "A global exception fault tolerance model for mpi," in *Workshop on Exascale MPI at Supercomputing Conference 2014*, 2014.

[44] M. M. Ali, J. Southern, P. Strazdins, and B. Harding, "Application level fault recovery: Using fault-tolerant open mpi in a pde solver," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 1169–1178.

[45] S. Pauli, M. Kohler, and P. Arbenz, "A fault tolerant implementation of multi-level monte carlo methods," *Parallel computing: Accelerating computational science and engineering (CSE)*, vol. 25, pp. 471–480, 2014.

[46] "Alsvid-uq." [Online]. Available: http://www.sam.math.ethz.ch/alsvid-uq/

[47] R. I. Van Der Wijngaart, M. R. U. Gamell, K. Teranishi, E. Valenzuela, M. A. Heroux, and M. R. U. Parashaar, "Fenix a portable flexible fault tolerance programming framework for mpi applications." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[48] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 895–906.

[49] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[50] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," *Acm sigplan notices*, vol. 47, no. 8, pp. 225–234, 2012.

[51] P. Du, P. Luszczek, and J. Dongarra, "High performance dense linear system solver with resilience to multiple soft errors," *Procedia Computer Science*, vol. 9, pp. 216–225, 2012.

[52] I. Rabinovitz, P. Shamis, R. L. Graham, N. Bloch, and G. Shainer, "Network offloaded hierarchical collectives using connectx-2's core-direct capabilities," in *European MPI Users' Group Meeting*. Springer, 2010, pp. 102–112.

[53] L. Wan, K. V. Mehta, S. A. Klasky, M. D. Wolf, H. Y. Wang, W. H. Wang, J. C. Li, and Z. Lin, "Data management challenges of exascale scientific simulations: A case study with the gyrokinetic toroidal code and adios," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2019.

[54] R. Löhner and J. D. Baum, "Load balancing for multiphysics," in *21st AIAA Computational Fluid Dynamics Conference*, 2013, p. 2856.

[55] G. J. Gassner and A. D. Beck, "On the accuracy of high-order discretizations for underresolved turbulence simulations," *Theoretical and Computational Fluid Dynamics*, vol. 27, no. 3-4, pp. 221–237, 2013.

[56] J. R. Bull and A. Jameson, "Simulation of the taylor–green vortex using high-order flux reconstruction schemes," *AIAA Journal*, vol. 53, no. 9, pp. 2750–2761, 2015.

[57] S. R. Ahmed, G. Ramm, and G. Faltin, "Some salient features of the time-averaged ground vehicle wake," *SAE Transactions*, pp. 473–503, 1984.

[58] M. Minguez, R. Pasquetti, and E. Serre, "High-order large-eddy simulation of flow over the "ahmed body" car model," *Physics of fluids*, vol. 20, no. 9, p. 095101, 2008.

[59] "Argo cluster, computing resource at george mason university." [Online]. Available: http://wiki.orc.gmu.edu/index.php/About_ARGO

[60] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future generation computer systems*, vol. 22, no. 3, pp. 303–312, 2006.

[61] A. Cavelan, J. Li, Y. Robert, and H. Sun, "When amdahl meets young/daly," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 203–212.

[62] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[63] A. Corrigan, F. Camelli, R. Löhner, and F. Mut, "Semi-automatic porting of a large-scale fortran cfd code to gpus," *International Journal for Numerical Methods in Fluids*, vol. 69, no. 2, pp. 314–331, 2012.

[64] M. Bahi and C. Eisenbeis, "High performance by exploiting information locality through reverse computing," in *2011 23rd International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2011, pp. 25–32.

[65] J. McCalpin, "Memory bandwidth and system balance in hpc systems," 2016. [Online]. Available: http://sc16.supercomputing.org/2016/10/07/sc16-invited-talk-spotlight-dr-john-d-mccalpin-presents-memory-bandwidth-system-balance-hpc-systems/index.html

[66] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, and B. Juurlink, "An evaluation of current simd programming models for c++," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, 2016, pp. 1–8.

[67] P. Estérie, J. Falcou, M. Gaunard, and J.-T. Lapresté, "Boost. simd: generic programming for portable simdization," in *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, 2014, pp. 1–8.

[68] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi, "Fault tolerance of mpi applications in exascale systems: The ulfm solution," *Future Generation Computer Systems*, vol. 106, pp. 467–481, 2020.

# Biography

Atis Degro received his Professional Bachelor degree in Civil Engineering from Riga Technical University in 2009. A year later he received a Professional Master degree from the same university. He went on to receive a Master of Science degree in Computational Mechanics at Technical University of Munich in 2012. After working for four years as research engineer at SL-Rash in Germany, he came to George Mason University for PhD studies. He graduated with a PhD degree in Physics from George Mason University in 2020.