Parallel, Dynamically Adaptive 2.5D Porous Media Flow Simulation on Xeon Phi Architectures

Master’s Thesis

Author: Aleksandra Atanasova Pachalieva
1st examiner: Prof. Dr. Michael Georg Bader
2nd examiner: Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Assistant advisors: Dipl. -Inf. (Univ.) Oliver Meister
M. Sc. Ao Mo-Hellenbrand
Thesis handed in on: March 15, 2016
I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

March 15, 2016

Aleksandra Atanasova Pchalieva
Abstract

sam(oa)$^2$ is a fast and memory efficient framework for the solution of PDEs based on Sierpinski curve traversals. The hybrid MPI+OpenMP parallelization of the code utilizes data parallelism in distributed and shared memory, which is a premise to achieve good performance on HPC systems. sam(oa)$^2$ contains a 2.5D porous media flow scenario suitable for vectorization with its regular grid in the vertical layers that makes it a good candidate for running on the Intel® Xeon Phi™ architecture. Further optimization techniques are proposed like vectorization, taking advantage of the 512-bit SIMD vector support, and load balancing, essential for the scalability due to the problem specific remeshing and varying computational load in symmetric mode. The performance of the framework is tested on the SuperMIC supercomputer using time-based load balancing.
First, I would like to thank my supervisors, M. Sc. Ao Mo-Hellenbrand and Dipl.-Inf. (Univ.) Oliver Meister, for the continuous support of my master’s thesis, for their patience, immense knowledge and understanding. They always assisted me when I was having questions no matter the day and time.

Also I would like to thank Univ.-Prof. Dr. Michael Georg Bader for giving me the opportunity to work on this topic and providing me with his wisdom.

Finally, I would like to thank my family and friends for their love and support, and Manuel M. Winkler, who never stopped encouraging me in my work.
# Contents

Abstract v

Contents ix

1 Introduction 1

2 sam(oa)$^2$— 2.5D Porous Media Flow 3
   2.1 Sierpinski Space-Filling Curve ............................................. 4
   2.2 Load Balancing in sam(oa)$^2$ Using Sierpinski Sections ................. 5
   2.3 2.5D Porous Media Flow ....................................................... 7

3 Intel® Xeon Phi™ Architecture on the SuperMIC System 9
   3.1 System Configurations .......................................................... 9
   3.2 Intel® Xeon Phi™ Execution Modes ........................................... 11
      3.2.1 Native Execution Mode ................................................... 11
      3.2.2 Offload Execution Mode ................................................. 12
      3.2.3 Symmetric Execution Mode .............................................. 13
   3.3 Performance Increasing Strategies for Intel® Xeon Phi™ Architecture  14
      3.3.1 Memory Performance ...................................................... 14
      3.3.2 Floating-Point Performance ............................................. 16
   3.4 Benefits of the Xeon Phi™ Coprocessors ................................... 16
   3.5 Effective Memory Bandwidth on Intel® Xeon Phi™ Measured by the STREAM Benchmark ...................................................... 17

4 Compilation and Execution of sam(oa)$^2$ on Intel® Xeon Phi™ in Symmetric Mode 21
   4.1 Modification of SConstruct .................................................... 21
   4.2 Compilation of sam(oa)$^2$ in Symmetric Mode ............................. 23
   4.3 Execution of sam(oa)$^2$ in Symmetric Mode ................................ 24

5 Roofline Model of the Pressure Linear Solver in sam(oa)$^2$ 27
   5.1 Operational Intensity of the Pressure Solver Kernel ....................... 28
   5.2 Roofline Model As a 2D Graph .............................................. 31

6 Performance Analysis on the SuperMIC System 33
   6.1 MPI Load Balancing .............................................................. 34
   6.2 Comparison Between Symmetric Mode, General-Purpose CPU and Co-
        processor Execution .............................................................. 37
Outline of the Thesis

Chapter 1: Introduction

Chapter 1 gives an introduction to the problem investigated within this master’s thesis. It mentions essential features of sam(oa)\(^2\) framework and Intel\(^\circledR\) Xeon Phi\(^\text{TM}\) architecture.

Chapter 2: sam(oa)\(^2\) – 2.5D Porous Media Flow

Chapter 2 presents the concept of sam(oa)\(^2\) framework with focus on the essential features for this thesis. The Sierpinski space-filling curve traversals and the load balancing algorithms implemented in sam(oa)\(^2\) are explained. This is followed by description of the 2.5D porous media flow scenario and its iterative workflow.

Chapter 3: Intel\(^\circledR\) Xeon Phi\(^\text{TM}\) Architecture on the SuperMIC System

Chapter 3 explains in detail the Intel\(^\circledR\) Xeon Phi\(^\text{TM}\) architecture running on the SuperMIC cluster and its system configurations and execution modes. The requirements for achieving good performance on the Intel\(^\circledR\) MIC architecture and the benefits of using Intel\(^\circledR\) Xeon Phi\(^\text{TM}\) are described. The last part of the chapter gives insights about the memory bandwidth of a SuperMIC node using the STREAM benchmark.

Chapter 4: Compilation and Execution of sam(oa)\(^2\) on Intel\(^\circledR\) Xeon Phi\(^\text{TM}\) in Symmetric Mode

Chapter 4 provides information on how to compile and run sam(oa)\(^2\) framework using Intel\(^\circledR\) Xeon Phi\(^\text{TM}\) in symmetric mode. All the code modifications and the needed scripts to fulfill the task are explained in detail.

Chapter 5: Roofline Model of the Pressure Linear Solver in sam(oa)\(^2\)

Chapter 5 describes the Roofline model analysis, which is later applied to the pressure linear solver of sam(oa)\(^2\). The Roofline model provides essential information about the performance expectations and bottlenecks.
Chapter 6: Performance Analysis on the SuperMIC System

Chapter 6 reveals several performance analysis tests and their corresponding results obtained on the SuperMIC system. The load balancing algorithms of sam(oa)\(^2\) are tested as well as the memory throughput of sam(oa)\(^2\) in comparison to the STREAM benchmark memory bandwidth. The execution of sam(oa)\(^2\) in symmetric mode is compared to other execution modes. In addition, the results from multiple nodes execution and hybrid MPI+OpenMP parallelization are analyzed.

Chapter 7: Optimization of sam(oa)\(^2\) for Efficiency

Chapter 7 includes Intel\(^\text{®}\) Xeon Phi\(^\text{™}\) specific strategies for optimization and vectorization of sam(oa)\(^2\) framework. They are implemented and tested on the 2.5D porous media flow scenario, aiming to increase its performance on the Intel\(^\text{®}\) MIC architecture.

Chapter 8: Conclusions

Chapter 8 shows the concluding remarks on the work and gives possible future directions for performance improvement of sam(oa)\(^2\) framework on Intel\(^\text{®}\) Xeon Phi\(^\text{™}\) architecture.
Chapter 1

Introduction

Nowadays increasing compute power allows engineers to examine physical problems with the help of simulations. These tools have been continuously developed in the last years and are established in various engineering and research fields. Simulations that capture the whole 3-dimensional space of a model are compute-intensive. However, in some cases it is possible to simplify models in order to reduce the computational effort.

One kind of these reducible problems are the so-called Shallow Water Equations that describe amongst others atmospheric, oceanic and subsurface flows. The mentioned equations are usually used for very large domains and the behavior of the vertical dimension can be captured with only a small number of layers [16]. Such 2.5D flow will be investigated within this work.

The 2.5D porous media flow scenario is analyzed as a particular application with the advantage of having a small vertical extend. The porous media flow is part of the sam(oa)$^2$ (Space-filling curves and Adaptive Meshes for Oceanic And Other Applications) [15] framework solving Partial Differential Equations (PDEs) on dynamically adaptive triangular grids. A special property of the corresponding 2.5D grid is its uniform data structure in the vertical dimension that supports a high degree of parallelization and vectorization. The grid is dynamically adaptive only in the horizontal dimension. These special properties make the 2.5D porous media flow a suitable scenario for the Intel® Xeon Phi™ architecture.

In the last decades, high-performance computing has gained importance and caused an intense development of new computational devices. One of these is the Intel® Xeon Phi™, which is particularly designed to solve highly parallel problems. After the announcement of Intel® ’s Many Integrated Cores (MIC) architecture in late 2012, the multi- and many-core computing has come into focus of interest of developers. The
The software industry has started to exploit multi-level parallelism instead of clock speed. The Intel® Xeon Phi™ coprocessors benefit from their large number of cores (up to 61 cores per card) and large SIMD vector instructions. This allows to run applications on up to 240 processes using shared and distributed memory parallelism. Intel® MIC architecture supports software developed in C, C++ and FORTRAN languages and allows applications to be built without changing their source code. This ensures fast portability, but it does not mean that the code will scale on the Intel® Xeon Phi™ system. To obtain good performance results, the application must utilize high degree of vectorization and parallelization. In addition, it must be specifically optimized for the Intel® MIC architecture.

The goal of this master’s thesis is to compile and run the 2.5D porous media flow problem on the Intel® Xeon Phi™ architecture. Furthermore, the potential of the sam(oa)² framework with regard to its performance on the many-core system will be analyzed and evaluated. Essential optimization strategies are implemented based on the Intel® Xeon Phi™ specifications, aiming to increase the vectorization and thus the performance.
Chapter 2

sam(oa)$^2$—2.5D Porous Media Flow

sam(oa)$^2$ (Space Filling Curves and Adaptive Meshes for Oceanic and Other Applications) [15] is a framework developed at the Technische Universität München for solving Partial Differential Equations (PDEs) using Sierpinski traversals on a dynamically adaptive triangular grid. It uses Sierpinski space-filling curves to provide memory and cache-efficient algorithms for grid generation, refinement and traversal. sam(oa)$^2$ supports Finite Element, Finite Volume and Discontinuous Galerkin discretizations. The framework is implemented in FORTRAN and combines memory efficiency with hybrid MPI+OpenMP parallelization.

In order to be memory efficient, the sam(oa)$^2$ software package requires Adaptive Mesh Refinement (AMR), in which some parts of the grid are represented more accurately than others using finer cells. The dynamically adaptive grid allows modifications of the number and size of the grid cells between time steps.

The hybrid MPI+OpenMP parallelization allows sam(oa)$^2$ framework to decrease memory usage per core and minimize execution time, taking advantage of the shared and distributed memory architecture. This allows simulating larger scenarios in less time. Furthermore, dynamic load balancing should be taken into account, due to the adaptive domain refinement and therefore, varying computational load for each core.

In this Chapter, we focus on the features of sam(oa)$^2$ essential for this thesis. For a comprehensive description of sam(oa)$^2$ refer to [15] and [16].
2.1 Sierpinski Space-Filling Curve

The Sierpinski space-filling curve [17] is defined as a recursive substructuring based on triangles. We can fill a square using a Sierpinski curve by dividing its diagonal into two isosceles right-angled triangles as a first step. Then each of these two elements can be refined recursively by splitting the hypotenuse until a sufficient grid resolution is obtained. Hanging nodes\(^1\) are not allowed. An example of a square domain with irregular grid is shown in Figure 2.1. The Sierpinski space-filling curve is the limit, obtained when the grid is refined infinitely.

![Figure 2.1: Sierpinski Space-Filling Curve in a Square Domain.](image)

The square domain is split into isosceles right-angled triangles with legs of equal size. The domain has irregular grid. The Sierpinski filling-curve is obtained, when the grid is refined infinitely. The line in red denotes the overall direction of the Sierpinski curve. The blue points separate the curve into sections with similar size. The boundaries of the sections are shown in different colors – red, blue, yellow and green.

The grid can be partitioned by dividing the space-filling curve into sections and assigning all triangles that are connected by one segment to one subdomain. The described sections are called Sierpinski sections and are defined as independent units. As one can see in the Figure 2.1, a section is built of multiple continuous cells implied by the Sierpinski order.

In the example grid from Figure 2.1, we have a domain divided in 20 triangles using 4 sections. The blue points in the figure indicate the separation points between the

\(^1\)Hanging node appears when one of the cells, sharing a node, is split and the other one is not.
sections. They divide the grid into chunks with similar size, which boundaries are depicted in different colors in Figure 2.1. In terms of the partition size, such a grid has a good quality. If a single part of the domain is refined recursively, while the rest of the domain is coarse, the partitioning will be unbalanced and will lead to inefficient parallelization.

In \textit{sam(oa)}^2, the Sierpinski space-filling curve provides a sequential order to iterate through all cells in the triangular grid. This order is used to arrange the sections of the grid for load balancing and parallelization.

2.2 Load Balancing in \textit{sam(oa)}^2 Using Sierpinski Sections

The Sierpinski order induced on the elements of the grid provides an algorithm for grid partitioning, traversal and refinement. \textit{sam(oa)}^2 obtains high parallelism and memory efficiency using load balancing algorithms based on Sierpinski sections. The load balancing algorithm is performed in each time step.

In the current Section 2.2, we introduce the basic partitioning algorithm, which includes adaptive refinement and load balancing. After that, we explain in detail the load balancing techniques implemented in \textit{sam(oa)}^2 framework and their advantages and disadvantages.

In \textit{sam(oa)}^2, the partitioning of the grid into sections is performed in the following steps:

1. \textit{Initial grid:} At the beginning of the simulation, the grid consists of a single section.

2. \textit{Adaptive refinement:} The grid is refined. During this step, the sections can be split or merged. Their size and location varies.

3. \textit{Load balancing:} The Sierpinski sections are redistributed between the processors to balance the load. Aiming to achieve a high level of parallelism, \textit{sam(oa)}^2 framework supports load balancing techniques based on the Chains-On-Chains partitioning [19]. During this step, a section cannot be divided or merged. All the cells that are processed by one process have to be continuous segments of the Sierpinski curve. Therefore, sections cannot be shifted arbitrarily between processes.

4. Go back to step 2 until the maximum grid depth is reached.
The total load $L$ is the sum over the load of each section as given in the formula below:

$$L = \sum_{i}^{s} l_i,$$

where $s$ denotes number of sections and $l_i$ is the load of a section. Then the optimal load per process $L_p$ is calculated as follows:

$$L_p = \frac{L}{n},$$

where $n$ is the number of processes.

$\text{sam}(oa)^2$ supports two load balancing algorithms – cell-based and time-based load balancing. Both of them are not initially designed to run on homogeneous hardware (like the Intel® Xeon Phi™ in symmetric mode). Nevertheless, the algorithms will be analyzed and tested on the MIC architecture within this thesis. Both load balancing techniques can be described as follows:

- **Cell-based load balancing:** In this load balancing algorithm, the load is distributed depending on the number of cells in the simulation. This algorithm will always balance the load in a static fashion without considering the heterogeneity of the system.

- **Time-based load balancing:** The load of each section is computed based on its computation time since the last remeshing. Depending on this time, the algorithm recalculates the load per process for the next iteration and may shift one or more sections to another process. On a homogeneous system with a static load per section, this algorithm converges immediately, otherwise, on a heterogeneous system, it becomes iterative. Depending on the time needed for the execution of one cell, the algorithm recalculates the number of cells per process for the next iteration. Even though the time-based load balancing algorithm is not designed for heterogeneous systems, it collects and accounts data that allows for treatment of the heterogeneity. This leads to slower convergence and higher execution time in the first time steps, but the distribution of the load between processes and the overall performance of the simulation is improved.

The initial assumption of the time-based load balancing algorithm is that the load of a section is independent of its location. However, this is not true for a heterogeneous system like Intel® Xeon Phi™ and therefore, several problems may occur. First, if one process is faster than the others, it might get overloaded
in the first time steps by receiving work from neighboring processes. This leads to the process running out of memory and thus, to a termination of the simulation. Second, if a fast process shifts a load to a slow process, this causes high barrier time\(^2\), because the load on the slow process is too much to execute. Both problems occur because a process cannot estimate how much time a Sierpinski section needs to be executed on a different process. To improve the time-based load balancing, we have to take into account the load location (host or processor), which will help the algorithm to distribute the load more equally between the heterogeneous processes.

Based on the nature of the presented algorithms, in the next Chapters, we will focus on the time-based load balancing algorithm. The cell-based load balancing is guaranteed to be inefficient on the Intel\(^\text{R}\) MIC architecture in symmetric mode, while the time-based load balancing might be able to resolve the heterogeneity of the system and balance the load between the processes.

2.3 2.5D Porous Media Flow

In sam(oa)\(^2\), the 2D triangular grid from Sections 2.1 and 2.2 can be extended into 2.5D prism grid. The framework does not have any restrictions on the number of Degree-Of-Freedom (DoF) saved in each entity (element, edge or vertex). Therefore, we can extend to 2.5D by storing DoF arrays instead of single values in each vertex. The DoF array represents the \(z\)-coordinate. The same approach is used for the cell data, where number of entities in the array is one fewer, because the cell data is located in the cell centers. In Figure 2.2, one can see a 2.5D domain with 4 layers (each vertex is associated with an array containing 5 entries, while each cell is associated with an array of size 4) [16]. A feature of this grid is that it can be dynamically refined or coarsened only in horizontal direction, while the vertical direction stays unchanged. This makes the 2.5D porous media flow scenario a good candidate to run on Intel\(^\text{R}\) Xeon Phi\(^\text{TM}\) architectures, because the kernels can be easily vectorized over the \(z\)-coordinate, when the number of layers is large enough.

In the following chapters, we denote the number of vertical layers as \(D\). Within sam(oa)\(^2\) framework it varies from 0 to 85 layers.

\(^2\)The barrier time represents the time needed for all the processes to reach a routine. If one process has already reached a barrier, it has to stop until the rest of the processes reach it as well.
The 2.5D porous media flow scenario executes the following steps in each iteration:

- **Computation:** During this step, the linear system is set and solved for the pressure. After that, the pressure gradients computed in the pressure solver are used to compute the time step size.

- . . .

- **Adaptive Refinement:** The grid is refined and the Sierpinski sections are split or merged.

- . . .

- **Load Balancing:** In the load balancing step, the grid is redistributed between the processes to ensure an appropriate load balancing.

The presented process workflow is simplified and the “. . .” items denote skipped steps, which are not relevant for this thesis. The complete workflow of the 2.5D porous media flow scenario can be found in [18]. The performance of the 2.5D flow depends mainly on the performance of the pressure linear solver and the rest of the computation step does not significantly affect the performance. The pressure solver is the most cumbersome part of the workflow and it is executed in each time step, which makes it a suitable code segment for further investigation and optimization.
Chapter 3

Intel® Xeon Phi™ Architecture on the SuperMIC System

The Intel® Xeon Phi™ coprocessor is based on the Intel® MIC architecture and combines many Intel® Pentium processor cores on a single chip. This allows the developers to port an existing C, C++ and FORTRAN code from the Intel® Xeon processor to the Intel® Xeon Phi™ coprocessor. The coprocessor is connected to an Intel® Xeon processor (referred to as host from here on) using the Peripheral Component Interconnect Express known as PCI Express bus.

In order to achieve full performance from the Intel® MIC architecture, applications must utilize high degree of parallelization and vectorization. The Intel® MIC architecture is specialized in highly parallel and vectorized code and it is not optimized in executing serial code. The speed and the efficiency of the Xeon Phi™ coprocessors come from its vector arithmetic units with 512-bit wide SIMD vector support and the new Intel® IMCI instruction set, capable of performing fused-multiply add instructions. The MIC architecture does not support any previous Intel® SIMD extensions like MME, SSE, SSE2, SSE3, SSE4.1, SSE4.2 or AVX instructions.

More details about the Intel® MIC architecture and the Intel® Xeon Phi™ coprocessors can be found in [2] or on the Intel® website [3].

3.1 System Configurations

SuperMUC at the Leibniz Supercomputing Centre is a Petascale supercomputer system located in Garching near Munich, Germany. During its last update, 32 new Intel® Xeon Phi™ nodes have been installed called SuperMIC. Each node in SuperMIC consists of
dual-socket Ivy Bridge host processor Xeon® E5-2650 and two Xeon Phi™ (MIC) coprocessors 5110P. The peak performance of one coprocessor is 1 TFlop in double precision (DP) and 2 TFlops in single precision (SP). The Intel® Xeon Phi™ coprocessor relies on the GDDR5 memory controllers for fast, high-bandwidth memory access. Its theoretical bandwidth is 320 GB/s with 8 GB of RAM memory. The 60 cores of the Xeon Phi™ 5110P coprocessor are interconnected by a bidirectional ring interconnect, while the Intel® Xeon® Ivy Bridge host is connected to the accelerators via PCI Express bus 2.0, limiting the bandwidth to 6.2 GB/s. The memory per node is in total 80 GB, where 64 GB are from the host and 16 GB (2 x 8 GB) from the two Xeon Phi™ cards. All Intel® Xeon Phi™ coprocessors can be accessed using a virtual bridge interface from the login nodes and from the compute nodes. An application can be compiled only on the login nodes (not on the coprocessors nor on the compute nodes). The operation system running on the Intel® Xeon Ivy Bridge and on the Intel® Xeon Phi™ coprocessors is SUSE Linux x86_64.

Further details about the SuperMIC system at the Leibniz Supercomputing Centre can be found in Table 3.1 and on their website [4]. For more information about Intel® Xeon Phi™ coprocessors refer to [5].

Table 3.1: System Configurations of the SuperMIC Cluster. *SuperMIC is the Intel® Xeon Phi™ cluster at the Leibniz Supercomputing Centre located in Garching near Munich, Germany. The data in this table is based on [1].*

<table>
<thead>
<tr>
<th>System overview</th>
<th>Intel® Xeon®</th>
<th>Intel® MIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Xeon® E5-2650</td>
<td>Xeon Phi™ 5110P</td>
</tr>
<tr>
<td>Cores/Clock Rate</td>
<td>2 x 8 / 2.6 GHz</td>
<td>60 / 1.053 GHz</td>
</tr>
<tr>
<td>Hardware Threads/Core</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Total Threads</td>
<td>32</td>
<td>240</td>
</tr>
<tr>
<td>SIMD Instruction Set</td>
<td>8 AVX</td>
<td>32 IMCI</td>
</tr>
<tr>
<td>SIMD Vector Length</td>
<td>256-bit</td>
<td>512-bit</td>
</tr>
<tr>
<td>Peak Performance (DP)</td>
<td>332.8 GFlops ~ 0.33 TFlops</td>
<td>1010.88 GFlops ~ 1.01 TFlops</td>
</tr>
<tr>
<td>Peak Bandwidth</td>
<td>2 x 59.7 GB/s</td>
<td>320 GB/s</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>Data Transfer with the Host</td>
<td>–</td>
<td>PCI Express bus 2.0 (6.2 GB/s)</td>
</tr>
</tbody>
</table>
3.2 Intel® Xeon Phi™ Execution Modes

Applications running on Intel® Xeon Phi™ architecture can be executed in native, offload, or symmetric mode depending on the code implementation and the used parallelism. Each of the described execution modes has its advantages and disadvantages. The three execution modes are described in the following subsections. Further details can be found in [6] and [7] or on the Intel® website [3].

3.2.1 Native Execution Mode

The native mode is the simplest way to run an application on the Intel® Xeon Phi™ coprocessor and this is also the natural first step of using Intel® MIC architecture. This mode is briefly depicted in Figure 3.1.

![Figure 3.1: Intel® Xeon Phi™ Native Execution Mode.](image)

To run in native mode, an application must be compiled on the host with the `-mmic` specific flag without additional changes in the source code. After that, the binary file is copied to the coprocessor and executed there. However, using the coprocessor in native mode restricts the code to only one coprocessor and neglects the resources of the host CPU. The memory limit of a coprocessor is approximately 8 GB and therefore, an execution in native mode performs good when the code has low memory needs and

...
intensive numerical computations. It is useful for performance testing and works well with interactive jobs.

3.2.2 Offload Execution Mode

When an application is executed in offload mode, the host starts the execution and initializes the data. After the initialization step, the application uses the PCI Express bus to send parts of the data and code blocks to be processed on the Xeon Phi™. The coprocessors send the results back to the host as soon as the data has been processed. In this mode, changes of the source code are necessary adding Intel® compiler’s offload directives or OpenMP 4.0 directives. They are used to indicate data and functions that should be offloaded to the Xeon Phi™ coprocessor and executed there. The Intel® compiler takes care of generating binary code for the coprocessor and the host, whenever it encounters an offload directive. Figure 3.2 shows an offload mode execution.

![Figure 3.2: Intel® Xeon Phi™ Offload Execution Mode. Offload mode execution uses the host to start and initialize the data of an application. Manually included offload directives indicate the data and the functions, which must be copied to the coprocessor and computed there (right: Offload Compute F1() and F2()). The communication between the host and the Xeon Phi™ coprocessor is carried out through the PCI Express bus.](image)

The host processor manages any high data and memory needs and at the same time, it sends expensive numerical segments to be computed on the Xeon Phi™ coprocessors. This mode requires the most programming effort. It allows a single code to use both the host and the coprocessor and avoids the challenging load balancing problems of
the symmetric mode, as described in Section 3.2.3. However, the main challenge of the offload mode is to achieve optimal workload for each core, in order to minimize the synchronization time. A disadvantage of the offload mode is that no MPI calls are allowed within an offload section.

3.2.3 Symmetric Execution Mode

The symmetric mode is a good choice, if the code is already designed to run on distributed memory systems using MPI protocol. The application is then executed simultaneously on the host and on the coprocessor, exploiting all the available resources. It is also the most flexible mode supporting 'any to any' messaging [14]. The application must be built twice, once for the Intel® Xeon Phi™ coprocessor, using the compiler flag \(-mmic\), and once for the host, using the \(-xHost\) compiling flag. Using symmetric mode over such a heterogeneous hardware is a source of challenging load balancing issues. Another bottleneck is the communication through the PCI Express bus between the processes executed on the host and on the coprocessors, which leads to high latency.

![Figure 3.3: Intel® Xeon Phi™ Symmetric Execution Mode. Symmetric mode execution uses all the resources of the host and the coprocessor. It runs simultaneously on the heterogeneous hardware using a MPI protocol, which divides the application into processes (Process 1 to 4). The source code must be built for both systems and copied to the appropriate locations. The processes running on the same system communicate fast, while the communication through the PCI Express bus (between the host (left) and the coprocessor (right)) may be a reason for high latency.](image-url)

Figure 3.3: Intel® Xeon Phi™ Symmetric Execution Mode. Symmetric mode execution uses all the resources of the host and the coprocessor. It runs simultaneously on the heterogeneous hardware using a MPI protocol, which divides the application into processes (Process 1 to 4). The source code must be built for both systems and copied to the appropriate locations. The processes running on the same system communicate fast, while the communication through the PCI Express bus (between the host (left) and the coprocessor (right)) may be a reason for high latency.
3.3 Performance Increasing Strategies for Intel® Xeon Phi™ Architecture

Writing and porting small projects to the Intel® MIC architecture is usually an easy task but obtaining good performance is an art. It requires not only deep knowledge of the source code and the language specifications but also understanding of the architecture. Within the current Section 3.3, an overview of the mainly used optimization strategies is given, which can be applied to achieve optimal performance on the Intel® Xeon Phi™ coprocessors [8] [9] [10]. The strategies are divided into two targets: memory performance and floating-point performance.

3.3.1 Memory Performance

The memory performance of an application is defined as the amount of reads from and writes to the memory (for the purpose of this master’s thesis, we count only the memory read accesses) and it is measured in Bytes per second (Byte/s). To increase the memory performance, we can use better data alignment and unit stride arrays with contiguous memory access, padding and prefetching, which are described as follows:

- **Data Alignment**: This is the way to create objects in memory with specific byte boundaries. Data alignment allows the processor to load and store chunks of data with higher efficiency. For the Intel® Xeon Phi™ coprocessor the optimal starting address for objects is modulo 64 Bytes (512 Bits) instead of 32 Bytes (256 Bits) for AVX and 16 Bytes (128 Bits) for SSE.

A key for good performance on the first generation of Intel® Xeon Phi™ coprocessors is to utilize vectorization. The usage of unaligned arrays triggers inefficient scalar instructions at the beginning or in the end of each array. We use the Intel® compiler flag `-align array64byte` to ensure that each array is aligned. However, in some cases, this flag is not sufficient and the programmer should add the Intel®-specific directive `!dir$ attributes align:64::A`, where variable A is declared. This directive tells the compiler explicitly to align the variable. In addition, the programmer should add the `!dir$ assume_aligned` directive, which tells the compiler for which loops the arrays can be accessed in aligned manner [11]. One can use Intel® compiler vector report options like `-qopt-report[=n]` to see, if the compiler is complaining and not aligning specific arrays (n indicates the level of detail in the report [12]). If the compiler is forced to align arrays, which are not aligned in reality, this leads to wrong results or segmentation faults.
Chapter 3. Intel® Xeon Phi™ Architecture on SuperMIC Supercomputing System

- **Multiversioned Arrays:** Multiversions of loops appear when the compiler does not know, if the pointers are pointing to contiguous chunks of memory or to an array section with a non-unit stride. In this case, the compiler generates separate code for each option and vectorizes the code only in case of contiguous memory (unit stride), but not otherwise. A solution for this problem is the usage of the FORTRAN 2008 feature – *contiguous* attribute [13]. This attribute can be declared for pointer or assumed shape array.

- **Padding:** This is a “rear-end” alignment technique [14]. Padding is used to increase the performance by allocating a suitable number of bit of unused data at the end of each row. The idea is to allocate enough space to guarantee that each element of a new row lies on the desired address boundary, which in our case is 64 Bytes. This is the length of the SIMD vector instructions. Padding will ensure that the compiler does not make any unaligned access to load data into the cache or the vector registers, which might cause performance loss.

   We want to guarantee that we use the full vector length without having peel loops or/and remainder loops. Peel loops are created when the memory access inside a loop is not aligned. The best way to solve this problem is to make sure that the compiler knows that the access is aligned (for example using the directive `!dir$ vector aligned`), which will guarantee that no peel loops are created. Remainder loops are generated when the number of loop iterations is not multiple of the vector length. To solve this problem, we can modify the source code so the remainder loop is not executed at runtime or by making the trip loop count larger, so that we decrease the overhead of the remainder loop execution. This can be achieved using the Intel® compiler flag `-unroll`.

   When the compiler flag `-opt-assume-safe-padding` is specified, the Intel® compiler assumes that variables and dynamically allocated memory are padded. This ensures that the code can access up to 64 Bytes more than what is defined in the program. The compiler does not add any padding for static and automatic objects, but anyway it assumes that the code can access up to 64 Bytes behind the end of each object. Therefore, we have to increase the size of the static and automatic objects, when we use this compiler flag.

- **Prefetching:** The Intel® Xeon Phi™ is an in-order architecture. For this reason, one should take care of the prefetching. Moving data from the memory to the math units can be delayed, when the mathematical operation (instruction) waits for the input data. To avoid this we can use a good prefetching strategy. It is recommended to leave the compiler to deal with the prefetching by setting the Intel® compiler flag `-opt/prefetch=n` with `n=1..4`, where a larger number
corresponds to a more aggressive prefetching by the compiler. When the compiler builds the application with optimization level -O2 or higher, then the prefetching flag is set to -opt-prefetch=3.

To ensure the prefetching, one can use prefetch directives or manual prefetching, but those are not going to be discussed or used in this thesis. For more details about prefetching refer to [14].

3.3.2 Floating-Point Performance

The floating-point performance of an application is defined by the number of floating point operations being executed per second on a given system. To increase it, we use vectorization, reduce branches or run more than one thread per core on Intel® Xeon Phi™ coprocessor. The options are described as follows:

- **Auto vectorization:** We can leave the Intel® compiler to automatically vectorize the code. The auto vectorization directives are enabled at default optimization level -O2 and even better performance is achieved using the -O3 compiler flag. This can be used for complicated loops, where it might break the data dependencies. The auto vectorization can be improved by adding other compiler options.

- **Xeon Phi™ specific vector instructions:** The speed and efficiency of the Intel® Xeon Phi™ comes from its vector units, therefore, vectorization is even more important than for Intel® Xeon®. The new instructions like gather/scatter, fused multiply-add, masked vector instructions etc. allow more loops to be vectorized on the MIC coprocessor than on the Intel® Xeon® host.

- **Directives to assist vectorization:** Directives like !dir$ simd and !dir$ ivdep are used to force vectorization in loops. They should be used carefully, because they are unsafe and often cause wrong results or segmentation faults, if they are applied wrongly.

- **Elemental functions and vector programming:** The usage of explicit vector programming and vector elemental functions is also beneficial for the auto vectorization of the code.

3.4 Benefits of the Xeon Phi™ Coprocessors

Intel® Xeon Phi™ coprocessors are designed very similar to the general-purpose CPUs. They support the programming models used for CPUs and are compatible with code
written in C, C++ or FORTRAN. The general-purpose CPU works on a high clock rate and it has a high single thread performance, while the coprocessors benefit from the large number of cores, working on low clock rate. Table 3.2 presents the main differences between the Intel® Xeon® E5-2650 processor and the Intel® Xeon Phi™ 5110P coprocessor.

<table>
<thead>
<tr>
<th>System overview</th>
<th>CPU Xeon® E5-2650</th>
<th>MIC Xeon Phi™ 5110P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>Clock Rate</td>
<td>2.6 GHz</td>
<td>1.053 GHz</td>
</tr>
<tr>
<td>Hardware Threads/Core</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>SIMD Width</td>
<td>256-bit wide</td>
<td>512-bit wide</td>
</tr>
<tr>
<td>DP GFLOPS/Core</td>
<td>∼ 21</td>
<td>∼ 16</td>
</tr>
</tbody>
</table>

Applications running on Xeon Phi™ coprocessors have to use multiple threads per core in order to achieve the theoretical peak performance of the system.

The Xeon Phi™ coprocessor executes applications using the already known hybrid MPI+OpenMP parallelization. Very often, optimization strategies for the Intel® MIC architecture are also beneficial for the general-purpose CPUs. The coprocessor focuses on high throughput using lots of weak threads and for most scenarios, it achieves two times better performance compared to the Xeon® E5 CPUs.

### 3.5 Effective Memory Bandwidth on Intel® Xeon Phi™

Measured by the STREAM Benchmark

Section 3.5 explains how to measure the effective memory bandwidth on a SuperMIC node using the STREAM benchmark. The benchmark result is used as a comparison to the memory throughput of the 2.5D porous media flow in Section 6.3, instead of the theoretical memory bandwidth of the whole SuperMIC system.

Aiming at better understanding of the Intel® Xeon Phi™ architecture and the capabilities of a SuperMIC node, we use the STREAM benchmark [32] to analyze the memory bandwidth of the system. Furthermore, based on [16] we expect that the pressure solver
kernel of \textit{sam(oa)}$^2$ is memory-bound, which means that the memory throughput is essential metric to characterize performance.

**Baseline Efficiency**

The STREAM benchmark is compiled and run with pure OpenMP parallelization. The source code of the benchmark is cross-compiled for the host (with \texttt{-xHost} flag) and for the Xeon Phi$^\text{TM}$ coprocessors (with \texttt{-mmic} flag) as follows:

**Listing 3.1: Compilation of the STREAM Benchmark Binary Files for Running on the SuperMIC node.** *We build the STREAM Benchmark executables for the Intel$^\text{R} \! \text{⃝} \! \text{Xeon Phi}^\text{TM} \! \text{architecture using the \texttt{-mmic} compiler flag. To obtain the total memory bandwidth of a single SuperMIC node, we run the benchmark independently on the host and on the two coprocessor cards. We use pure OpenMP parallelization.}*

\begin{verbatim}
  icc -xHost -O3 -openmp -mcmodel=medium stream_omp.c -o stream_omp.host
  icc -mmic -O3 -openmp -mcmodel=medium stream_omp.c -o stream_omp.mic
\end{verbatim}

The benchmark uses an array size of 310 M elements, which corresponds to approximately 7 GB of memory. This is the maximum number of elements, which we are allowed to use on the Intel$^\text{R} \! \text{⃝} \! \text{Xeon Phi}^\text{TM}$ coprocessor before the STREAM benchmark caches due to exceeding the memory limits of the system. In addition, we have to use the flag \texttt{-mcmodel=medium}, which tells the compiler that the size of the data used by the application is larger than 2 GB. The compilation commands above generate two binary files – one for the host (\texttt{stream_omp.host}) and one for the coprocessors (\texttt{stream_omp.mic}). The STREAM benchmark does not support distributed memory parallelization, thus the benchmark has to be run separately on the host and on each of the coprocessors.

The STREAM benchmark is executed on one SuperMIC node using 32 OpenMP threads on the Intel$^\text{R} \! \text{⃝} \! \text{Xeon}$ host and 59 OpenMP threads on each of the Intel$^\text{R} \! \text{⃝} \! \text{Xeon Phi}^\text{TM}$ coprocessors. The total number of OpenMP threads used for the node is 150 threads. The results of the STREAM benchmark can be seen in Table 3.3.

The estimation of the memory bandwidth in the STREAM benchmark and the memory throughput in \textit{sam(oa)}$^2$ differs in two aspects. The first aspect is that the memory throughput in \textit{sam(oa)}$^2$ is calculated by adding the memory accesses required for all vertex and cell data. The data is assumed to be touched at least once during traversal. We count an access per memory location only once, which means that only a read operation is counted. In contrast, the STREAM benchmark counts the read and write accesses. Therefore, to have consistent results, we divide the memory bandwidth from the benchmark by two before inserting it in Table 3.3.
Table 3.3: STREAM Benchmark Memory Bandwidth on a SuperMIC Node. The tests are executed on a SuperMIC node, which contains one host and two Xeon Phi\textsuperscript{TM} cards. The total memory bandwidth is the sum of all the computing units of the node.

<table>
<thead>
<tr>
<th>System</th>
<th># Threads</th>
<th>Memory Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>32</td>
<td>41.687 GB/s</td>
</tr>
<tr>
<td>MIC</td>
<td>59</td>
<td>2 x 53.775 GB/s</td>
</tr>
<tr>
<td>Total Memory Bandwidth</td>
<td></td>
<td>149.237 GB/s</td>
</tr>
</tbody>
</table>

The second aspect is that the pressure kernel operations in sam(oa)\textsuperscript{2} do not exactly reflect the STREAM benchmark kernels. The SCALE kernel of the STREAM benchmark [32] has the most similarities to the linear pressure solver from the 2.5D porous media flow scenario. The SCALE kernel is defined as follows:

\[ a(i) = q \cdot b(i), \quad (3.1) \]

where \(a(i)\) and \(b(i)\) denote arrays with index \(i\), and \(q\) is a scalar value. In the SCALE kernel, the multiplication result of \(q\) and \(b(i)\) is saved in a new array \(a(i)\). In contrast, the resulting value of the multiplication of the pressure kernel in sam(oa)\textsuperscript{2} is assigned to the same variable as the one used for the operation. The kernel is rather similar to:

\[ a(i) = q \cdot a(i), \quad (3.2) \]

This implies that the optimal memory bandwidth measured by the STREAM benchmark is slightly higher than the given one in Table 3.3.
Chapter 4

Compilation and Execution of sam(oa)$^2$ on Intel® Xeon Phi™ in Symmetric Mode

In this Chapter, we will discuss how to build and execute sam(oa)$^2$ on the Intel® Xeon Phi™ coprocessors. Due to the size of the sam(oa)$^2$ project, this task is cumbersome and should not be underestimated. For generating the building script, the SCons construction tool is used [20]. We choose to apply the symmetric mode execution because the application will take advantage of the entire computational power of the system, while minimizing the code changes. More information on how to port applications for the Intel® Xeon Phi™ in symmetric mode can be found in [21].

4.1 Modification of SConstruct

sam(oa)$^2$ utilizes the SCons tool for compilation considering it automates the complicated compilation process using Python scripts. The SConstruct is the build file of this construction tool. To execute sam(oa)$^2$ in symmetric mode, it should be modified to generate binaries for the host and for the coprocessors. Hence, 'mic' variable should be added, which specifies the Intel® Xeon Phi™ target machine:
Listing 4.1: Definition of 'mic' Variable in the SConstruct File. An additional value ('mic') for the machine variable in the SConstruct file is defined. This allows the user to build applications for the Intel® MIC architecture by setting machine='mic' in the Python script.

```python
EnumVariable('machine', 'target machine', 'host',
    allowed_values=('SSE4.2', 'AVX', 'host', 'mic'))
```

Now, we can include all the flags we want to attach for the compiler by using the machine variable. Listing 4.1 shows how this is implemented into the SConstruct file.

Listing 4.2: Implementation of Machine Specific Compiler Flags. In the SConstruct file, we define different compiler flags for different machines. Here one can see the added compiler flags for the target machine set to 'host' and to 'mic'.

```python
if env['compiler'] == 'intel':
    if env['machine'] == 'host':
        env['F90FLAGS'] += ' -xHost'
    elif env['machine'] == 'mic':
        env['F90'] += ' -mmic'
        env['LINK'] += ' -mmic'
```

If Intel® compiler is used, we can check, if the target machine is 'mic' and then attach -mmic to the compiler flags.

Another small change to the SConstruct file is commenting out some of the compiling and linking flags used before:

Listing 4.3: Commented Flags Unsuitable for the 2.5D Porous Meda Flow Scenario. Some of the compiler flags used for the 2D porous media flow have to be commented out, because they are not suitable for it.

```python
if env['compiler'] == 'intel':
    # env['F90FLAGS'] += ' -fast -fno-alias -align all -inline-level=2 -funroll-loops
    # -unroll -no-inline-min-size -no-inline-max-size
    # -no-inline-max-per-routine -no-inline-max-per-compile
    # -no-inline-factor -no-inline-max-total-size'
    env['F90FLAGS'] += ' -g -fast -fno-alias -align all -inline-level=2 -funroll-loops -unroll'
    # env['LINKFLAGS'] += ' -O3 -ip -ipo'
```

Originally, the 2.5D porous media flow scenario is designed to be 2D. In the 2D case, the pressure solver is simpler than in the 2.5D porous media flow. For this reason, it has been more time-consuming to call the solver than to execute it and the usage of inlining flags has been beneficial for the performance. Since we work with the 2.5D porous media flow, we do not have the less cumbersome pressure
Chapter 4. Compilation and Execution of sam(oa)$^2$ on Intel® Xeon Phi™ in Symmetric Mode

solver any longer and inlining large structures of code will only slow down the execution of the application. It is also likely that the pressure solver kernels do not fit anymore in a single instruction cache. Therefore, we have to take away some compiler flags as shown in Listing 4.3.

Last modification of the SConstruct file is to add .host and .mic extensions to the file names of the host and coprocessor binaries, respectively. This modification does not change the executables but it ensures that we do not overwrite or interchange them. The implementation is given in Listing 4.4.

Listing 4.4: .host and .mic Extensions Added to the Binary Files Names. The last modification in the SConstruct file is to add .host and .mic extensions to the binary files of the host and coprocessor, respectively. They are used to guarantee that the binary files will not be interchanged or overwritten.

```python
if env['machine'] == 'host':
    program_name += '.host'

if env['machine'] == 'mic':
    program_name += '.mic'
```

4.2 Compilation of sam(oa)$^2$ in Symmetric Mode

With the modification of the SConstruct file, the compilation for the two systems, i.e., host and MIC, can be done with the help of two Python scripts listed in Listing 4.5. The two Python scripts build a 2.5D porous media flow scenario with 8 vertical layers. The script on the left side compiles the executable for the host and the one on the right side compiles the executable for the Xeon Phi™ coprocessors. The variable machine specifies the system on which the application will be executed.

Listing 4.5: Python Configuration Scripts for Building sam(oa)$^2$ Executables. The Python files are saved as porous_media_flow_8l_host.py and porous_media_flow_8l_mic.py for the host and for the Intel® Xeon Phi™ coprocessors, respectively. Their variables can be modified depending on the desired configuration.

```python
flux_solver="upwind"    flux_solver="upwind"
compiler="intel"        compiler="intel"
asagi="False"          asagi="False"
openmp="noomp"         openmp="noomp"
mpi="default"          mpi="default"
machine="host"         machine="mic"
layers="8"              layers="8"
```
The Python scripts from Listing 4.5 are saved as `porous_media_flow_8l_host.py` and `porous_media_flow_8l_mic.py` for the host and for the Intel® Xeon Phi™ coprocessors, respectively. For more information on how to define the Python scripts and build different scenarios with sam(oa)$^2$ framework, refer to [23].

The executables for the host and for the Intel® Xeon Phi™ coprocessors are compiled as follows:

**Listing 4.6: Compilation of sam(oa)$^2$ in Symmetric Mode Using the SCons Python Scripts.** The 2.5D porous media flow scenario is build in symmetric mode using the previously described Python scripts.

```
scons config=../path_to_file/porous_media_flow_8l_host.py
scons config=../path_to_file/porous_media_flow_8l_mic.py
```

In case that the application uses external data or external libraries, which are not already installed on the SuperMIC, the user should make sure that they are also built with `-mmic` compiler flag. They have to be copied and linked properly to the rest of the application as shown in [24].

### 4.3 Execution of sam(oa)$^2$ in Symmetric Mode

There are two approaches to start an application in symmetric mode on the SuperMIC cluster. The first option is to use an interactive job, and the second one is to submit a batch job.

After starting an interactive job, we reserve a node for a fixed amount of time. To start a simulation, we login to the host and copy the binary files to the `/scratch/` directory. Then we manually run the executables in symmetric mode using the command:

**Listing 4.7: Simulation Started Manually in Symmetric Mode Using an MPI Command.** The `mpiexec` command is used for submitting a simulation in symmetric mode on the SuperMIC after submitting an interactive job. Depending on the needs, we can set different number of MPI ranks and arguments.

```
mpiexec -host hostname -ib -n 16 ./bin/executable_name.host [arguments]:
- host hostname-mic0 -n 118 /scratch/executable_name.mic [arguments]:
- host hostname-mic1 -n 118 /scratch/executable_name.mic [arguments]
```

One drawback of using an interactive job is that the simulation is limited to be executed only on one node. Another drawback is that the user has to copy
manually the binary files, the external data and all the needed libraries. This can be very time-consuming and inefficient, when we aim to execute a large number of simulations.

The second approach is to use a batch script, which internally reserves a given number of nodes and copies automatically all the binary files, data and libraries. An example of the batch script can be found on the Leibniz Supercomputing Centre website [4]. Listing 4.8 shows a sample script that is used to run sam(oa)\textsuperscript{2} in symmetric mode on the SuperMIC.

**Listing 4.8: A Batch Script Used to Execute sam(oa)\textsuperscript{2} Simulations in Symmetric Mode.** This batch script is used for running applications on the SuperMIC system. The number of nodes can be increased up to 32 nodes. The executables, the data and the libraries are automatically set or copied. This script is based on an example batch script from [4].

```bash
#!/bin/bash
#@ wall_clock_limit = 01:00:00
#@ job_name = test
#@ job_type = parallel
#@ class = phi
#@ node = 1
#@ node_usage = not_shared
#@ initialdir = $(home)/path/to/samoa
#@ output = test-$jobid.out
#@ error = test-$jobid.out
#@ notification=always
#@ notify_user=name@mytum.de
#@ queue

export I_MPI_DAPL_PROVIDER_LIST=ofa-v2-mlx4_0-1u
export I_MPI_MIC=enable

output=./output
bin_mic_path=./bin/executable_name.mic
bin_mic=executable_name.mic
dir_mic=/scratch
bin_host=./bin/executable_name.host
arg=""
debug=""

taskspermic=118
tasksperhost=16

command="mpiexec"

for i in `cat $LOADL_HOSTFILE`; do
    numhosts=$((numhosts+1))
    host=`echo $i | cut -d- -f1`;
    hosts="$hosts $host";
    mic0=$host-mic0
    mic1=$host-mic1
    mics="$mics $mic0 $mic1"
```
In the batch file, we specify the maximum wall-clock time for the simulation, the job name, type, class and the number of nodes. Furthermore, we set the path to the sam(oa)$^2$ framework in our file system. After the basic specifications of the script, two MPI variables are exported. The I_MPI_MIC enables MPI tasks to run natively on the Intel® Xeon Phi™ coprocessors, while the I_MPI_DAPL_PROVIDER_LIST variable sets the DAPL provider. After that, the location of the executables is given and the arguments for the simulation are set. The number of MPI ranks running on the coprocessors and on the host are defined in the variables taskpermic and taskperhost, respectively. We use the mpiexec command to run the simulation. In the loop, the executables and the external data are copied to the coprocessors. Then the MPI command is assembled automatically, depending on the number of nodes and the arguments. Finally, the names of the hosts and coprocessors are printed and the command is executed. More details about the structure of the batch file can be found in [25].
Chapter 5

Roofline Model of the Pressure Linear Solver in sam(oa)$^2$

In the past, improving the CPU performance was achieved by increasing the core frequency. About a decade ago, this strategy reached its physical limit. In order to improve the performance further, the complexity of the CPUs has been increased. Hence, nowadays it is harder to understand the complex CPU behavior and to compare different systems. Therefore, there exists a need for an intuitive and simple model. The Roofline model [26] [27] offers this simplicity, and can be applied to a large number of current and future computer architectures, even to heterogeneous multicore computers like the Intel® Xeon Phi™.

The Roofline model provides the programmer not only with realistic performance expectations, but also it enumerates the potential performance bottlenecks. This can help to find weak spots in the software and to implement particular optimization strategies. Additional material regarding performance tuning of scientific applications using the Roofline model can be found in [28].

Before proceeding with the Roofline model analysis, it is necessary to clarify some important concepts like communication, operational intensity and machine balance.

Communication

Communication is the transfer of data from the memory to the processors. It is limited by the computer architecture and the processor-memory interconnection. The Intel® Xeon Phi™ coprocessors are connected to the Ivy Bridge host through a PCI Express bus, which bounds the memory bandwidth to 6.2 GB/s.
Operational Intensity

Operational intensity represents the ratio between operations per Byte and DRAM traffic [26] or how many operations per Byte are executed in regard to memory traffic. The operational intensity is measured in Flops per Byte (Flops/Byte).

Machine Balance

Machine balance is similar to the operational intensity but it measures the ratio between the peak floating-point performance and the peak memory bandwidth of a system. It is measured in Flops/Word, where one word is assumed to be 8 Byte long. The machine balance represents how many floating-point operations a computer can perform with a word, before the next word is fetched. High machine balance value indicates an unbalanced machine. If the processor cannot execute enough floating-point operations on the fetched word, then it will stay idle, while it waits for the next word to arrive.

5.1 Operational Intensity of the Pressure Solver Kernel

Within this work, we want to find the operational intensity and to apply the Roofline model to the pressure solver kernel of the 2.5D porous media flow. To find the operational intensity of a kernel, we have to calculate the memory traffic and the instructions count.

The kernel of the pressure solver is an algorithm for solving linear systems of equations using the Jacobi-preconditioned Conjugate Gradient method. The following are the core functions executed iteratively.

- `pre_dof_op()`
- `apply3D()`
- `get_trace3D()`
- `post_dof_op()`
- `reduce_dof_op()`

These functions are part of the sam(oa)² framework and they are located either in the Darcy_local_function_spaces.f90 or in the PipeCG.f90 file, which can be found
in the sam(oa)² framework repository [23]. We will not go into details about the implementation of the pressure linear solver. For more information refer to [18].

To calculate the operational intensity of the pressure solver kernel, we analyze the memory traffic by counting the read and write accesses of each kernel variables – matrix $A$, right hand side $\text{rhs}$ and unknown variable of the linear solver equation $x$, direction vector $d$, vector $v$ and residual $r$. In addition, we take into account the size of each array. All the information is summarized in Table 5.1.

**Table 5.1: Memory Traffic of the Pressure Linear Solver.** The memory traffic of the pressure solver is a sum of all the read and write accesses to the used variables for solving the linear system. The dimension of the used arrays is also taken into account.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Memory Access</th>
<th>Dimensions</th>
<th>Traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix $A$</td>
<td>read</td>
<td>$(7, D)$</td>
<td>$1 \cdot 7 \cdot D$</td>
</tr>
<tr>
<td>Right hand side $\text{rhs}$</td>
<td>read</td>
<td>$(D+1)$</td>
<td>$1 \cdot (D + 1)$</td>
</tr>
<tr>
<td>Unknown $x$</td>
<td>read and write</td>
<td>$(D+1)$</td>
<td>$2 \cdot (D + 1)$</td>
</tr>
<tr>
<td>Direction vector $d$</td>
<td>read and write</td>
<td>$(D+1)$</td>
<td>$2 \cdot (D + 1)$</td>
</tr>
<tr>
<td>$v = A \cdot d$</td>
<td>read and write</td>
<td>$(D+1)$</td>
<td>$2 \cdot (D + 1)$</td>
</tr>
<tr>
<td>Residual $r$</td>
<td>read and write</td>
<td>$(D+1)$</td>
<td>$2 \cdot (D + 1)$</td>
</tr>
</tbody>
</table>

**Memory Traffic:** $7 \cdot D + 9 \cdot (D + 1)$

In Table 5.1, we calculate the memory traffic depending on the number of the vertical layers $D$. In order to calculate the total memory traffic, we multiply the traffic of matrix $A$ by the number of cells and the traffic of the other vectors by the number of vertices in the simulation as shown in the next equation:

$$\text{Total Memory Traffic} = 7 \cdot D \cdot \text{cells} + 9 \cdot (D + 1) \cdot \text{vertices}. \quad (5.1)$$

In the 2.5D porous media flow scenario, the number of cells in the grid is approximated by the number of vertices as follows:

$$\text{cells} \approx 2 \cdot \text{vertices}. \quad (5.2)$$
We reformulate the equation of the total memory traffic as given in Equation 5.3:

\[
\text{Total Memory Traffic} \approx 7 \cdot D \cdot 2 \cdot \text{vertices} + 9 \cdot (D + 1) \cdot \text{vertices} \\
\approx (23 \cdot D + 9) \cdot \text{vertices}.
\]  

(5.3)

For calculating the number of double instructions in the pressure solver kernel, we count the number of double floating point instructions executed in each of the pressure solver functions. Table 5.2 shows the type and the number of double instructions per function.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Instruction Type</th>
<th>Number of Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre_dof_op()</td>
<td>3 multiplications</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 additions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 subtraction</td>
<td>6 \cdot D \text{ Flops}</td>
</tr>
<tr>
<td>apply3D()</td>
<td>14 multiplications</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14 additions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14 subtractions</td>
<td>42 \cdot D \text{ Flops}</td>
</tr>
<tr>
<td>get_trace3D()</td>
<td>14 additions</td>
<td>14 \cdot D \text{ Flops}</td>
</tr>
<tr>
<td>post_dof_op()</td>
<td>1 division</td>
<td>1 \cdot D \text{ Flops}</td>
</tr>
<tr>
<td>reduce_dof_op()</td>
<td>7 multiplications</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 additions</td>
<td>11 \cdot D \text{ Flops}</td>
</tr>
</tbody>
</table>

Number of Instructions of the Pressure Solver: \(74 \cdot D \text{ Flops}\)

The number of instructions depends, just like the total memory traffic, on the number of vertices. Therefore, the total number of double instructions is estimated by the following function:

\[
\text{Double Instructions} = 74 \cdot D \cdot \text{vertices} \text{ [Flops]}.
\]  

(5.4)
To calculate the operational intensity of the pressure solver, we use the following formula, combining the total memory traffic and the number of double instructions:

\[
\text{Operational Intensity} \left[ \frac{\text{Flops}}{\text{Byte}} \right] = \frac{\text{Double Instructions}}{\text{Total Memory Traffic}} \left[ \frac{\text{Flops}}{8 \text{ Bytes}} \right]
\]

\[
= \frac{\text{Double Instructions}}{\text{Total Memory Traffic} \cdot 8} \left[ \frac{\text{Flops}}{\text{Byte}} \right]
\]

\[
\approx \frac{74 \cdot D \cdot \text{vertices}}{(23 \cdot D + 9) \cdot \text{vertices} \cdot 8} \left[ \frac{\text{Flops}}{\text{Byte}} \right]
\]

\[
\approx \frac{74 \cdot D}{(23 \cdot D + 9) \cdot 8} \left[ \frac{\text{Flops}}{\text{Byte}} \right]
\]

\[
\Rightarrow \text{Operational Intensity} \left[ \frac{\text{Flops}}{\text{Byte}} \right] \leq \frac{2}{5} \left[ \frac{\text{Flops}}{\text{Byte}} \right]
\]

Depending on the number of vertical layers \( D \), the operational intensity varies from 0.29 for \( D=1 \) to 0.40 for \( D=85 \), which is the maximum allowed value for the vertical layers of the 2.5D porous media flow.

### 5.2 Roofline Model As a 2D Graph

The Roofline model combines the theoretical peak performance, the memory bandwidth and the operational intensity in one 2D graph, shown in Figure 5.1. The theoretical peak performance of the Intel® Xeon Phi™ is 1 TFlop and the memory bandwidth is 320 GB/s. Figure 5.1 outlines the Roofline model for an Intel® Xeon Phi™ card. The x-axis describes the operational intensity in Flops per Byte. The y-axis is the attainable floating-point performance in GFlops per second. The peak floating point performance represents the hardware limit of the system. Therefore, no higher values can occur during the kernel execution. The peak memory bandwidth draws another system limit, which cannot be exceeded. The intersection point of the peak memory bandwidth and peak floating-point performance is called ridge point [26]. All the kernels with operational intensity left from the ridge point are memory-bound (hitting the slanted part of the roof), while all the kernels with operational intensity right from the ridge point are compute-bound.
(hitting the flat part of the roof). In the Roofline model for the Intel® Xeon Phi™ coprocessor card, the ridge point is far on the right, which means that achieving peak performance is a difficult task and only kernels with very high operational intensity can reach peak performance.

![Graph](image)

**Figure 5.1: Roofline Model of the Intel® Xeon Phi™ Coprocessor and the Pressure Solver Kernel As a 2D Graph.** Roofline model of the Intel® Xeon Phi™ coprocessor card with peak memory bandwidth and floating-point performance in blue. The operational intensity of the pressure solver kernel varies in the range [0.29, 0.40] depending on the number of vertical layers $D$, shown as a gray trapezoid.

As mentioned before, the operational intensity of the pressure solver kernel varies from 0.29 to 0.40 depending on the number of vertical layers of the 2.5D porous media flow scenario. The red line defines the minimum pressure solver operational intensity, while the green line defines the maximum pressure solver operational intensity. From the Roofline model, we can guarantee that the pressure solver kernel is memory-bound.

In order to increase the operational intensity of the pressure solver kernel, we can improve the vectorization, apply SIMD instructions, ensure data alignment and prefetching.
Chapter 6

Performance Analysis on the SuperMIC System

The performance of the sam(oa)$^2$ framework is analyzed on SuperMUC, a Petascale system located in Garching near Munich, Germany. For the purpose of this master's thesis, we use the Many Cores nodes also called SuperMIC. SuperMIC is a cluster with 32 nodes each having an Intel® Xeon® Ivy-Bridge E5-2650 host with 16 cores and two Intel® Xeon Phi$^\text{TM}$ 5110P coprocessors with 120 cores in total. The presented tests are performed using Intel® Fortran Compiler 16.0 and Intel® MPI 5.1, both installed on the SuperMIC cluster. We use double precision arithmetics. For more details about the SuperMUC system installed at the Leibniz Supercomputing Centre in Garching near Munich, Germany, refer to their website [30].

In this Chapter, several performance analysis results are described. In Section 6.1, we investigate the time-based and cell-based MPI load balancing algorithms and their behavior on a heterogeneous hardware system like SuperMIC. In Section 6.2, we compare three execution modes – symmetric mode using simultaneously the host and the coprocessors, host mode using only the Ivy Bridge host and coprocessors mode using only the Intel® Xeon Phi$^\text{TM}$ coprocessors. Similar experiment can be found in [31]. In Section 6.3, the 2.5D flow scenario is executed on a single node with increasing number of vertical layers using pure MPI parallelization. The memory throughput of sam(oa)$^2$ is compared to the available memory bandwidth on a SuperMIC node, measured with the STREAM benchmark. In Section 6.4, tests with hybrid MPI+OpenMP parallelization are performed. In Section 6.5, we analyze the performance of sam(oa)$^2$ running on multiple nodes with weak scaling.
Chapter 6. Performance Analysis on the SuperMIC System

6.1 MPI Load Balancing

Running simulations on a heterogeneous hardware such as the symmetric mode of the Intel® Xeon Phi™ coprocessors often leads to load balancing problems. The processors executing a simulation simultaneously, differ in their properties like clock rate, memory, SIMD instructions, memory bandwidth and peak performance as shown in Chapter 3. Therefore, we want to test how the load balancing strategies of sam(oa)$^2$ deal with the heterogeneity of the hardware.

For this comparison, we decided to use the load balancing algorithms implemented in the sam(oa)$^2$ framework – cell-based load balancing and time-based load balancing, explained in detail in Section 2.2. Both algorithms are not initially designed to run on heterogeneous hardware. For this reason, we investigate their behavior on the Intel® Xeon Phi™ architecture.

Figure 6.1 shows how both of the algorithms balance the workload of a 2.5D porous media flow simulation after the first 10 time steps, which is equivalent to one phase. On the x-axis, the two load balancing algorithms are given – cell-based and time-based. The y-axis represents the number of cells processed by one host core and one coprocessor core. The cores of the coprocessors are depicted in blue color, while the cores of the host are depicted in red. The simulation is executed using 236 cores per coprocessor and 16 cores per host.

On one hand, by using the cell-based load balancing, the amount of cells executed on the host and on the coprocessors is almost the same. One host core executes about 1000 cells more than a coprocessor core, which is a negligible difference. On the other hand, the time-based load balancing algorithm allows the host cores to run about 15 times more cells than the coprocessor cores. One cell processed on the host is much cheaper, than one cell on the coprocessor. This can be explained by the heterogeneous hardware, for example, the Intel® Xeon® Ivy Bridge host works on 2.6 GHz clock rate, while the Intel® Xeon Phi™ coprocessor works only on 1.1 GHz clock rate. In addition, the sam(oa)$^2$ framework is optimized to run on general-purpose CPUs, not on MIC architecture. Both systems – Intel® Xeon® Ivy Bridge host and Intel® Xeon Phi™ coprocessors, differ further in their architecture, which contributes to the gap between the host core and the coprocessor core in Figure 6.1.

Nevertheless, the strength of the Intel® Xeon Phi™ is not in the single core execution, but in using all the 60 cores per card. In the Table 6.1, one can see
Chapter 6. Performance Analysis on the SuperMIC System

Figure 6.1: Comparison of the Load Between the Host and the Coprocessor Based on the Load Balancing Algorithm. Two simulations are executed using different load balancing algorithms – time-based and cell-based. The cell-based load balancing separates the load between the host and coprocessor cores almost equally. In contrast, the time-based algorithm balances the load depending on its execution time. Therefore, it sends around 15 times more cells to the Ivy Bridge host, which runs on higher clock frequency and calculates a cell faster. The time-based algorithm tackles the heterogeneity of the Intel® Xeon Phi™ symmetric mode and it is used for the following tests.

Table 6.1: Total Number of Cells Calculated on the Host and on the Coprocessors Using Time-based Load Balancing. The total number of the calculated cells per system – host or coprocessor, is given. We consider the number of cores per system and the respective number of cells executed per core. The Intel® Xeon Phi™ coprocessors compute two times more cells in total than the Ivy Bridge host.

<table>
<thead>
<tr>
<th>System</th>
<th>Cores</th>
<th>Cells executed per core</th>
<th>Total number of cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>16 x 64</td>
<td>64 599</td>
<td>1 033 584</td>
</tr>
<tr>
<td>Coprocessors</td>
<td>2 x 236</td>
<td>4297</td>
<td>2 028 184</td>
</tr>
</tbody>
</table>

The Intel® MIC coprocessors perform approximately 2 M cells, while the host executes only 1 M cells.

that even though one core of the coprocessors executes much less work than the Ivy Bridge host, the total number of processed cells still dominates.
In order to achieve these results, we executed simulations with different configurations and we observed an interesting trend. The simulations running with time-based load balancing spend drastically more time in MPI barrier (up to 80% of the total time). Most probably the time-based load balancing algorithm has not converged yet and it needs more time steps to start splitting the work in appropriate loads. This load balancing algorithm is not designed to tackle the bottlenecks of a heterogeneous hardware. However, it collects and analyzes data that allows threatening the problems caused by heterogeneity. This often leads to slower convergence of the time-based algorithm and imbalanced load between the host and the coprocessors during the initial phase. During the first time steps of the simulation, the Ivy Bridge host is overloaded but this will balance out in time. By increasing the number of time steps, we allow the time-based load balancing algorithm to recognize its error and reduce the load of the Ivy Bridge cores. Therefore, all the following simulations are executed using 10 times more time steps (100 time steps). Figure 6.2 shows that after the 3rd time step the number of cells per process stays stable. The displayed values are only from one process, running on the host.

![Figure 6.2: Number of Cells Executed on an Ivy Bridge Host Process. Due to the time-based load balancing, there are large fluctuations of the number of cells in the first time steps. After the 3rd time step the algorithm converges and the number of cells saturates.](image-url)
The saturation of the number of steps after the 3\textsuperscript{rd} time step means that the time-based load balancing algorithm has converged. For the future measurements, we skip the grid setup and its initialization, because in the first 10 regular time steps the grid is refined and distributed until it is fully resolved up to an error indicator. Instead, we use the results from the last 10 time steps (referred as a phase in the next sections).

Time-based load balancing splits the work between the cores of the heterogeneous system much more adequate, than the cell-based technique. Therefore, we chose to use the time-based load balancing for all of the following tests.

### 6.2 Comparison Between Symmetric Mode, General-Purpose CPU and Coprocessor Execution

An application running on the MIC architecture can be executed in native, offload or symmetric mode, as shown in Section 3.2. Within this work, the symmetric mode is used, because it allows distributed memory parallelism with MPI calls and in addition, it utilizes the whole system – host and coprocessors. We want to perform a direct performance comparison between a symmetric execution, general-purpose CPU execution (Intel\textsuperscript{R} Xeon\textsuperscript{R} E5-2650 Ivy Bridge host), and coprocessor execution (Intel\textsuperscript{R} Xeon Phi\textsuperscript{TM} 5110P coprocessor) based on [31]. All the systems use pure MPI parallelization. The three test execution modes are described as follows:

1. **Symmetric execution:** The symmetric mode is simultaneously running MPI ranks on the host and on the coprocessor. It takes advantage of the whole SuperMIC node and executes the code using 16 MPI ranks on the host and 118 MPI ranks for each of the two Intel\textsuperscript{R} Xeon Phi\textsuperscript{TM} cards, which results in 252 MPI ranks in total. The memory available on the system is 80 GB with 64 GB on the host and 16 GB (2 x 8 GB) on the coprocessors.

2. **General-purpose CPU execution:** The application is running only on the Intel\textsuperscript{R} Xeon\textsuperscript{R} Ivy Bridge host using 16 MPI ranks. The memory available for the execution is 64 GB. During this execution the coprocessors in the node are idle.

3. **Coprocessor execution:** This execution mode is almost identical to the native execution from Section 3.2. We run the application using the symmetric execution mode command specifying the number of the MPI ranks
on the host to be zero. The total number of MPI ranks is 236 ranks in this case (2 x 118 ranks) and the available memory is 16 GB (2 x 8 GB of memory located on the coprocessors).

Table 6.2 gives an overview of the configuration of the tested systems.

Table 6.2: Configurations of the Test Systems. We compare the following system executions – symmetric, general-purpose CPU (host) and coprocessor execution. The number of MPI ranks for each system is given, as well as the available memory for the execution.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Avail.</th>
<th>MPI Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory</td>
<td>Host</td>
</tr>
<tr>
<td>Symmetric execution</td>
<td>80 GB</td>
<td>16</td>
</tr>
<tr>
<td>General-purpose CPU execution</td>
<td>64 GB</td>
<td>16</td>
</tr>
<tr>
<td>Coprocessor execution</td>
<td>16 GB</td>
<td>–</td>
</tr>
</tbody>
</table>

Hyper-threading is not used in full capacity for Intel® Xeon® and Intel® Xeon Phi™, which are capable of running 2 hyper-threads per core (total of 32 threads) and 4 hyper-threads per core (total of 240 threads), respectively. Hyper-threading allows additional threads to be executed, but does not provide faster access to data or larger number of instructions. Hence, it has rarely shown any improvement of the performance for the used test case.

The 2.5D porous media flow scenario is chosen to be large enough for sufficient utilization of the available cores and also to fit to the memory of the Intel® Xeon Phi™. The simulation is executed with 2 M cells. Table 6.3 shows the phase time of the 2.5D porous media flow with $D=8$ vertical layers and 4 Sierpinski sections per process running on the previously described execution types – symmetric, general-purpose CPU and coprocessor. The wall-clock time is measured for the last phase of the simulation.

The symmetric execution outperforms the other two configurations, even though the used time-based load balancing algorithm is not designed for heterogeneous hardware. One phase in symmetric mode takes only 753.3 seconds, while the general-purpose CPU execution needs almost twice the time – 1399.4 seconds. This large difference in the execution time suggests that sam(oa)$^2$ benefits most of the symmetric execution mode and gives as good performance as possible, without modifying the framework. Running sam(oa)$^2$ in symmetric mode allows to use the entire computational power of the system.
Table 6.3: Execution Time Measured for the Test Systems. The wall-clock time is measured for the symmetric, general-purpose CPU (host) and coprocessor executions. The tests are run using 2.5D porous media flow scenario with $2M$ elements. The symmetric execution performs two times better than the other two (low number is good).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Phase Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric execution</td>
<td>753.3 seconds</td>
</tr>
<tr>
<td>General-purpose CPU execution</td>
<td>1399.4 seconds</td>
</tr>
<tr>
<td>Coprocessor execution</td>
<td>1237.3 seconds</td>
</tr>
</tbody>
</table>

To show the full potential of the symmetric mode, we will perform further tests on the Intel® Xeon Phi™ architecture. We investigate the performance and memory throughput based on the number of vertical layers of the scenario. This is described in Section 6.3.

6.3 Evaluation of Single Node Performance for Increasing Number of Vertical Layers

The 2.5D porous media flow problem is very computationally expensive, when the number of layers $D$ is low. If the number of layers is larger, the pressure solver kernel allows better vectorization and the overhead per element gets smaller. In Section 6.3, we concentrate on the memory throughput and the execution time of the sam(oa)$^2$ framework for increasing number of vertical layers.

6.3.1 Memory Throughput of sam(oa)$^2$ in Comparison to the STREAM Benchmark

Based on the results from the Roofline model described in Chapter 5, the pressure solver is expected to be memory-bound on the analyzed system. A node in the SuperMIC has a theoretical peak performance of $2354.56 \text{ GFlop/s}^1$ in double precision and memory bandwidth of $759.4 \text{ GB/s}^2$. Both values are calculated as a sum:

$^1$Theoretical Peak Performance of a SuperMIC Node:
$$2 \cdot 1010.88 \text{ [GFlops]} + 332.8 \text{ [GFlops]} = 2354.56 \text{ [GFlops]}.$$  

$^2$Theoretical Memory Bandwidth of a SuperMIC Node:
$$2 \cdot 320 \text{ [GB/s]} + 119.4 \text{ [GB/s]} = 759.4 \text{ [GB/s]}.$$
of the theoretical peak performance or the memory throughput of two Intel® Xeon Phi™ cards and a Xeon® Ivy Bridge host. Thus, to attain peak performance at least 49.6 double precision operations per memory fetch should be executed. Even in extreme case only 6 operations per fetch are executed in our 2.5D porous media flow scenario [16]. If the executed framework cannot perform the number of flops per memory fetch, then the CPU has to stay idle until the next data is fetched from memory. Hence, the performance is clearly limited by the memory bandwidth and we concentrate in investigating the memory throughput of \( \text{sam(oa)}^2 \) depending on the number of vertical layers.

Figure 6.3 shows the measured memory throughput of \( \text{sam(oa)}^2 \) with different number of layers (\( D=1 \) to \( D=32 \) layers). The simulations are executed using 16 MPI ranks on the host and 118 MPI ranks on each coprocessor (the total number of ranks for a node is 252). The 2.5D porous media flow has 2 M elements and uses 4 Sierpinski sections per process.

![Figure 6.3: Memory Throughput Obtained on a SuperMIC Node Using from 1 to 32 Vertical Layers. The memory throughput of \( \text{sam(oa)}^2 \) is measured for the 2.5D porous media flow scenario with 2M cells using 1 to 32 vertical layers on a single SuperMIC node. The memory throughput is given in GB per second and reaches its highest value using 8 layers – 51.75 GB/s, after that it saturates (high number is good).](image-url)
The memory throughput for one node reaches 51.75 GB/s for \( D=8 \) layers and saturates after that. The lower memory throughput with 16 layers is not significant, but it may be explained by the computational difference between the executed scenarios. Even though only the number of layers is changed, actually each of the scenarios differs in its execution, because the number of coupling iterations or linear solver iterations is decided at runtime and can be much lower, or higher, for a particular simulation.

We measure the memory bandwidth of a SuperMIC node with the STREAM benchmark, as explained in Section 3.5, to compare it to the memory throughput of the \( \text{sam(oa)}^2 \) framework obtained on a single node. Table 6.4 shows the memory bandwidth ratio between the STREAM benchmark and \( \text{sam(oa)}^2 \).

### Table 6.4: Memory Throughput of \( \text{sam(oa)}^2 \) Compared to the STREAM Benchmark.

The memory throughput of \( \text{sam(oa)}^2 \) is measured for 2M elements using 1 to 32 layers on a single SuperMIC node compared to the STREAM benchmark. As the number of layers is increased, the memory throughput increases from 14\% to 34\% and saturates after 8 layers.

<table>
<thead>
<tr>
<th>Framework</th>
<th>1 layer</th>
<th>4 layers</th>
<th>8 layers</th>
<th>16 layers</th>
<th>32 layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM benchmark</td>
<td>149.24 (100%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{sam(oa)}^2 )</td>
<td>21.14</td>
<td>38.17</td>
<td>51.75</td>
<td>46.70</td>
<td>52.03</td>
</tr>
<tr>
<td>Ratio</td>
<td>14.17%</td>
<td>25.58%</td>
<td>34.68%</td>
<td>31.29%</td>
<td>34.86%</td>
</tr>
</tbody>
</table>

The \( \text{sam(oa)}^2 \) framework reaches only 34\% memory throughput of the STREAM benchmark and saturates after \( D=8 \) layers. The memory bandwidth of the SuperMIC node is very high and therefore, hard to reach for frameworks, which are not designed or modified to run on Intel\textsuperscript{®} Xeon Phi\textsuperscript{TM} architecture. In addition, the communication and data transfer between the Ivy Bridge host and the Intel\textsuperscript{®} Xeon Phi\textsuperscript{TM} coprocessors is limited to 6.2 GB/s by the PCI Express bus 2.0. This limitation can lead to high latency and low memory throughput.

#### 6.3.2 Comparison of the Execution Time for Increasing Number of Vertical Layers

The performance of \( \text{sam(oa)}^2 \) is characterized also by the wall-clock time needed for the framework to execute one phase. Such a comparison gives insights on how the scenario scales when the number of layers increases.
Figure 6.4 depicts the amount of time that $\text{sam(oa)}^2$ needs to execute one phase of the simulation depending on the number of vertical layers.

![Figure 6.4: Execution Time Obtained on a SuperMIC Node Using from 1 to 32 Vertical Layers.](image)

The phase time of $\text{sam(oa)}^2$ is measured for the 2.5D porous media flow scenario using $2M$ elements with 1 to 32 vertical layers. The scenario performs good with 8 or more layers. The improved performance comes from the higher number of vertical layers that allow better vectorization.

The scenario with 8 vertical layers performs much better than the ones using 1 and 4 layers. The Intel® compiler takes advantage of the larger number of vertical layers and vectorizes parts of the code. The $\text{sam(oa)}^2$ framework needs 753 seconds to execute the 8 layers scenario and almost doubles every time we double the number of layers. The scenarios using 16 and 32 layers perform good, however, there is no significant improvement concerning the execution time.

### 6.4 Hybrid MPI+OpenMP Performance

Hybrid MPI+OpenMP parallelization is performed on a SuperMIC node, using one MPI rank per Intel® Xeon Phi™ coprocessor and Ivy Bridge host. Each MPI rank has 236 OpenMP threads. We expect from the hybrid parallelization
to decrease the memory per core and reduce the execution time, exploiting the shared and distributed memory parallelism. This will allow us not only to execute simulations faster, but also to run larger simulations on the SuperMIC nodes.

The 2.5D flow scenario running with hybrid MPI+OpenMP parallelization has 8 vertical layers and 4 Sierpinski sections per process. The time-based load balancing algorithm is used. As a comparison a pure MPI parallelization running in symmetric mode with 118 MPI ranks per coprocessor and 16 MPI ranks per host is executed.

Table 6.5 shows a comparison of the memory throughput between the pure MPI and the hybrid MPI+OpenMP parallelization.

**Table 6.5: Comparison of the Memory Throughput Between Executions Using Pure MPI and Hybrid Parallelization.** The memory throughput is measured on the 2.5D porous media flow scenario with 8 vertical layers. Pure MPI and hybrid MPI+OpenMP parallelizations are compared, where the MPI execution performs five times better with respect to the memory throughput (high number is good).

<table>
<thead>
<tr>
<th>Parallelization</th>
<th>Memory Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure MPI</td>
<td>50.4 GB/s</td>
</tr>
<tr>
<td>Hybrid MPI+OpenMP</td>
<td>10.5 GB/s</td>
</tr>
</tbody>
</table>

The hybrid parallelization has almost five times lower memory throughput running on the same system as the pure MPI parallelization. In Figure 6.5, we plot the elapsed CPU time for the components (computation time, synchronization time and time spend at barrier) of both simulations to find the reason for the bad performance of the hybrid parallelization. The load balancing time and the (de)allocation time are negligible, therefore, they are not shown in the figure.

The total CPU time of the execution with hybrid parallelization is 37 times larger than the pure MPI execution. The computation time for pure MPI dominates during the run, while in the hybrid version the application spends most of the time in synchronization. In Table 6.6, the time spent for each component is represented as a percentage of the total CPU time.

In the pure MPI parallelization, the computation time is 68.79% of the total time, while in the hybrid parallelization, the component occupies only 3.56%. The hybrid MPI+OpenMP parallelization spends most of the time in synchronization and barrier time. In addition, the load balancing time is also dramatically increased.
Chapter 6. *Performance Analysis on the SuperMIC System*

Figure 6.5: Comparison Between Pure MPI and Hybrid MPI+OpenMP Parallelization Based on the Execution Time of the Components. The total execution time is broken down into the following components – computation time, synchronization time, time spent at barrier and load balancing time. The (de)allocation time is not shown in the figure, because its value is negligible. The hybrid execution needs about 37 times more time to be executed than the pure MPI. The main reason for this is the large amount of time spent in synchronization (low number is good).

Table 6.6: Percentage Representation of the Components Time to the Total CPU Time. In the pure MPI parallelization, the computation time is 68.79% of the total time, while in the hybrid parallelization the component takes only 3.56%. The hybrid MPI+OpenMP parallelization spends most of the time in synchronization and barrier time. In addition, the load balancing time is also dramatically increased.

<table>
<thead>
<tr>
<th>Parallelization</th>
<th>Computation</th>
<th>Synchronization</th>
<th>Barrier</th>
<th>Load Balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure MPI</td>
<td>68.79%</td>
<td>14.92%</td>
<td>14.40%</td>
<td>0.03%</td>
</tr>
<tr>
<td>Hybrid MPI+OpenMP</td>
<td>3.56%</td>
<td>78.64%</td>
<td>8.13%</td>
<td>2.19%</td>
</tr>
</tbody>
</table>

The number of the MPI ranks is only 3 and therefore, the barrier time stays low in comparison to the synchronization time. We have high global communication and imbalance of the processes. The exact reasons for this problem have to be investigated further.
6.5 Multi-Node Performance

After analyzing the potential of a single SuperMIC node using sam(oa)$^2$ framework, we are interested in its performance on multiple nodes. In the following Section 6.5 this will be investigated.

The distributed memory performance test is a weak scaling study performed with the 2.5D porous media flow scenario using 1 to 8 nodes on the SuperMIC cluster. The tests are executed with 2 M to 16 M elements and the number of MPI ranks per node is 252. To obtain larger number of elements, the refinement depth of the 2.5D porous media flow is increased from 20 to 23 maximum levels.

The scenario running on 2 nodes executes successfully, but by increasing further the number of nodes, the simulation gets terminated after several phases. The sam(oa)$^2$ output file states that the process has been interrupted. The reason for the termination is the time-based load balancing strategy. Based on the heterogeneous hardware of the MIC architecture, one of the processes is faster than the others and it gets overloaded with work. Hence, this process runs out of memory and the simulation terminates. In the time-based load balancing algorithm, the available memory on the cores is not considered and the amount of allocated data per node is nonuniformly distributed. For example, when we run a scenario on 4 nodes, one of the nodes can get 30-40% of the total load, which will lead to out-of-memory problems. Nevertheless, all of the tested scenarios complete their 3rd phase, which is shown in the following figures.

The memory throughput of the sam(oa)$^2$ 2.5D porous media flow scenario running on multiple nodes is depicted in Figure 6.6. The memory throughput slightly increases from 1 node to 2 nodes execution with approximately 5.6 GB/s. However, this trend does not continue with doubling the number of nodes. The memory throughput stays stable at around 50-55 GB/s, even though the number of cells per scenario and the number of nodes doubles. Based on the results from the Roofline model shown in Chapter 5, the performance is clearly memory-bound, although compared to the STREAM benchmark memory bandwidth, from Section 3.5, the memory throughput of sam(oa)$^2$ hardly reaches 35%. Based on the results, we conclude that the memory bandwidth of the system is very high and hard to reach. The sam(oa)$^2$ framework performs as good as possible, but it does not scale well due to load imbalance. Figure 6.7 shows that the number of cells per process based on the time-based load balancing has fluctuations even in the 5th phase. This indicates that the algorithm is not converged yet and the Ivy Bridge host
Figure 6.6: Memory Throughput of sam(oa)\(^2\) Measured on Multiple Nodes. The memory throughput is measured for simulations running on 1 to 8 nodes using weak scaling. The value of the memory throughput does not increase as expected with the size of the problem or the number of nodes (high number is good).

is overloaded. The load change leads to exceeding of the memory limits for one of the cores and the simulation terminates. Figure 6.7 displays data from the simulation running on 4 nodes, but the same holds for the execution on 8 nodes.

To investigate further how the load fluctuations affect the overall performance, we show the elapsed CPU time of the simulations and its components — computation, synchronization, barrier and load balancing in Figure 6.8. In the bottom part of Figure 6.8, the same data is represented as a percentage of the total CPU time.

Figure 6.8 shows that the execution time of the simulation running on more than one node is dominated by the barrier time. From 50% to 70% of the total CPU time is spent in MPI barriers, where multiple cores are staying idle waiting for other cores to finish with their work.

We have to change the time-based load balancing algorithm to obtain better results on multiple nodes. In the currently used algorithm, it is not considered how much time a load takes to be executed on the other processors, but only on its own. In the time-based load balancing algorithm, the load of a Sierpinski section is assumed to be independent of its location. This is not fulfilled anymore when
Figure 6.7: Number of Cells Executed on a Ivy Bridge Host Using Time-based Load Balancing. The large fluctuations in the number of cells per host process is due to the time-based load balancing. After the 56-th time step the simulation terminates, because of overloading one of the processes. Therefore, it is not guaranteed that the load balancing algorithm has converged even after 5 phases.

we are using a heterogeneous hardware. This means, that a fast process can get overloaded with work, because one cell takes less time to be processed. After that, the process will try to distribute some of its load to other processes. If a fast process sends even a small load to a slow process, the load might be too large for the slow process, which leads to larger barrier and synchronization time. These problems arise because the load is a relative abstraction depending on its location – host or coprocessor. To obtain better results, we have to improve the time-based load balancing algorithm or implement a new one, which can tackle better the heterogeneity of the system, regardless of the number of used nodes or problem size. This task is beyond the scope of this master’s thesis and it should be further investigated.
Figure 6.8: Comparison Between Multi-Node Execution from 1 to 4 SuperMIC Nodes. The total execution time is broken down into three different components – computation time, synchronization time, time spent at barrier and load balancing. Several components are not shown in the figure, because their execution time is negligible. The figure above shows the CPU time in seconds, while the figure below gives the respective data in percentage of the total time. The barrier time of the executions using more than one node, dominates spending from 50% to 70% of the total time. The load-balancing algorithm in those cases is not converged yet and the work imbalance causes large barrier time values.
Chapter 7

Optimization of sam(oa)² for Efficiency

To achieve good performance on the Intel® Xeon Phi™ architecture is a challenging task. It depends not only on the suitable vectorization and parallelization of the source code, but also on the computational complexity of the executed algorithms.

In the following Chapter, the main optimization strategies are explained, implemented and tested in the context of the sam(oa)² framework. Section 7.1 reveals details about the pressure solver kernel, which is encapsulated and tested as a stand alone program. Several optimized versions of the code are implemented and executed using OpenMP parallelization. Their wall-clock time is compared and the version with the best potential is applied to the sam(oa)² framework. Section 7.2 shows the code transformations in sam(oa)² framework. Section 7.3 presents the performance results of the optimized pressure kernel tested on a single SuperMIC node.

7.1 Optimization of the Pressure Solver Kernel

The pressure solver is the most expensive memory-bound part of the 2.5D porous media flow scenario in sam(oa)². Despite the low operational intensity of the pressure kernel, we want to optimize the kernel for the Intel® MIC architecture. We know that the code block is memory-bound based on our findings from the Roofline model in Chapter 5. The structure of the whole pressure solver algorithm has to be reviewed to achieve significant performance improvement. Large changes
in the pressure solver implementation would require multiple changes in other parts of the framework. Altogether, this improvement strategy is very time-consuming and goes beyond the scope of this thesis.

In addition, such modifications might not be beneficial for the other scenarios like 2D porous media flow and tsunami wave propagation. Therefore, only vectorization and data alignment strategies are applied within this Chapter with the aim of introducing as little changes to the framework as possible.

For the purposes of Section 7.1, the most expensive part of the pressure kernel is extracted from the code and prepared to be executed as a small stand-alone program. This allows to test the optimization strategies easier and faster. Later on, they can be implemented in the sam(oa)$^2$ framework. To make a performance analysis of the program, we use random numbers for data initialization for all experiments described in the current Section 7.1. The program is called test$_{apply3D}()$ and is displayed in Listing 7.1.

**Listing 7.1: test$_{apply3D}()$ Program.** In the program test$_{apply3D}()$, the data is declared and initialized. The function makes a call to the apply$_{3D}()$ subroutine within a parallel OpenMP loop with static scheduling. The wall-clock time is measured for the execution of this loop.

```fortran
program test_apply3D
  use OMP_LIB
  use apply

  real start, finish
  real :: x1(D+1), x2(D+1), x3(D+1)
  real :: r1(D+1), r2(D+1), r3(D+1)
  real :: A(D, 7)

  ! initialize the vectors x1, x2, x3, r1, r2, r3, A
  ! using random numbers
  call random_number(x1)
  call random_number(x2)
  call random_number(x3)
  call random_number(A)

  call cpu_time(start)
  !$OMP PARALLEL DO SCHEDULE (STATIC, ITER/118)
  do i = 1, ITER
    call apply3D(x1, x2, x3, r1, r2, r3, A)
  end do
  !$OMP END PARALLEL DO
  call cpu_time(finish)

  print*, "time to compute", ITER-1,
      "iterations of apply3D: ",
      finish-start, "seconds"
end program
```

In the main program `test_apply3D()`, the used arrays are declared and initialized with random numbers. After that, a parallel OpenMP loop is executed using static scheduling between the OpenMP threads and the subroutine `apply3D()` is called within this loop. `apply3D()` can be seen in Listing 7.2. The code is simplified for readability.

**Listing 7.2: apply3D() Subroutine.** The most expensive part of the pressure solver is the `apply3D()` subroutine. The code sample is formulated in a vector-friendly manner and it is additionally simplified for readability. The size of the 1D arrays `r1`, `r2`, `r3`, `x1`, `x2`, `x3` is `(D+1)`, and the size of the 2D array `A` is `(7, D).

```
module apply
  integer, parameter :: D = 85
  integer, parameter :: ITER = 1180000
contains
  subroutine apply3D(x1, x2, x3, r1, r2, r3, A)
    real, intent (in) :: x1 (:), x2 (:), x3(:)
    real, intent (inout) :: r1 (:), r2 (:), r3 (:)
    real, intent (in) :: A (:, :)

    /bottom horizontal contributions
    r1 (1: D) = r1 (1: D) + A(:, 1) * (x1 (1: D) - x2 (1: D))
    r2 (1: D) = r2 (1: D) + A(:, 1) * (x2 (1: D) - x1 (1: D))
    r3 (1: D) = r3 (1: D) + A(:, 2) * (x3 (1: D) - x2 (1: D))
    r2 (1: D) = r2 (1: D) + A(:, 2) * (x2 (1: D) - x3 (1: D))

    /vertical contributions
    r1 (1: D) = r1 (1: D) + A(:, 3) * (x1 (1: D) - x1 (2: D +1))
    r1 (2: D +1) = r1 (2: D +1) + A(:, 3) * (x1 (2: D +1) - x1 (1: D))
    r2 (1: D) = r2 (1: D) + A(:, 4) * (x2 (1: D) - x2 (2: D +1))
    r2 (2: D +1) = r2 (2: D +1) + A(:, 4) * (x2 (2: D +1) - x2 (1: D))
    r3 (1: D) = r3 (1: D) + A(:, 5) * (x3 (1: D) - x3 (2: D +1))
    r3 (2: D +1) = r3 (2: D +1) + A(:, 5) * (x3 (2: D +1) - x3 (1: D))

    /top horizontal contributions
    r1 (2: D +1) = r1 (2: D +1) + A(:, 6) * (x1 (2: D +1) - x2 (2: D +1))
    r2 (2: D +1) = r2 (2: D +1) + A(:, 6) * (x2 (2: D +1) - x1 (2: D +1))
    r3 (2: D +1) = r3 (2: D +1) + A(:, 7) * (x3 (2: D +1) - x2 (2: D +1))
    r2 (2: D +1) = r2 (2: D +1) + A(:, 7) * (x2 (2: D +1) - x3 (2: D +1))
  end subroutine
end module
```
7.1.1 Optimized Version 1: Contiguous Arrays

A feature of FORTRAN 2008 [13] is the contiguous attribute, which ensures that the function accesses a contiguous chunk of memory. This allows the compiler to vectorize the code. The apply3D() subroutine is modified as given in Listing 7.3.

Listing 7.3: Contiguous Attribute Applied to the Used Vectors. The arrays r1, r2, r3, x1, x2, x3, A are declared as contiguous, which allows better vectorization.

```fortran
real, contiguous, intent(in) :: x1(:,), x2(:,), x3(:)
real, contiguous, intent(inout) :: r1(:,), r2(:,), r3(:)
real, contiguous, intent(in) :: A(:, :)
```

7.1.2 Optimized Version 2: Data Alignment

To ensure that the compiler allocates aligned memory, we use the Intel® directive `!dir$ attributes align:64` for all 1D arrays, as shown in Listing 7.4.

Listing 7.4: Attributes Align Directive. In the main function test_apply3D(), where the arrays r1, r2, r3, x1, x2, x3 are declared, we use the FORTRAN directive `!dir$ attributes align:64` to make the compiler allocate the variables in aligned fashion.

```fortran
real :: x1(D+1), x2(D+1), x3(D+1)
real :: r1(D+1), r2(D+1), r3(D+1)
real :: A(D, 7)
!dir$ attributes align:64 :: x1, x2, x3, r1, r2, r3
```

However, this directive is not sufficient to generate aligned loops and therefore, we should add the `!dir$ assume_aligned var:64` directive, which tells the compiler which arrays can have an aligned access. One can see how this is implemented in Listing 7.5.

Listing 7.5: Assume Aligned Directive. In the apply3D() subroutine, we add the `!dir$ assume_aligned var:64` directive, which tells the compiler which arrays can have an aligned access.

```fortran
real, contiguous, intent(in) :: x1(:,), x2(:,), x3(:)
real, contiguous, intent(inout) :: r1(:,), r2(:,), r3(:)
real, contiguous, intent(in) :: A(:, :)
!dir$ assume_aligned x1:64, x2:64, x3:64, r1:64, r2:64, r3:64
```
Chapter 7. Optimization of \textit{sam(oa)}^2 for Efficiency

7.1.3 Optimized Version 3.1: SIMD Instructions

To assist the vectorization in the loops from the \texttt{apply3D()} function in Listing 7.6, we add the \texttt{!dir$ simd} directive. This directive does not check whether the vectorization will be beneficial for the performance. The compiler tries to vectorize the code regardless of the outcome. Inappropriate usage of the FORTRAN \texttt{!dir$ simd} directive leads to wrong results or segmentation faults [29].

Our first approach of using SIMD directives can be seen in Listing 7.6.

\textbf{Listing 7.6: SIMD Directive.} In the \texttt{apply3D()} function, we insert the \texttt{!dir$ simd} directive before each FORTRAN loop, which forces vectorization, without performing any safety checks. The programmer should apply this directive carefully, because its inappropriate usage leads to wrong results or segmentation faults.

```
subroutine apply3D(x1, x2, x3, r1, r2, r3, A)
  real, contiguous, intent(in) :: x1(:), x2(:), x3(:)
  real, contiguous, intent(inout) :: r1(:), r2(:), r3(:)
  real, contiguous, intent(in) :: A(:, :)
  ! dir$ assume_aligned x1:64, x2:64, x3:64, r1:64, r2:64, r3:64

  /bottom horizontal contributions
  !dir$ simd
  r1(1:D) = r1(1:D) + A(:,1) * (x1(1:D) - x2(1:D))
  !dir$ simd
  r2(1:D) = r2(1:D) + A(:,1) * (x2(1:D) - x1(1:D))
  !dir$ simd
  r3(1:D) = r3(1:D) + A(:,2) * (x3(1:D) - x2(1:D))
  !dir$ simd
  r2(1:D) = r2(1:D) + A(:,2) * (x2(1:D) - x3(1:D))

  /vertical contributions
  !dir$ simd
  r1(1:D) = r1(1:D) + A(:,3) * (x1(1:D) - x1(2:D+1))
  !dir$ simd
  r1(2:D+1) = r1(2:D+1) + A(:,3) * (x1(2:D+1) - x1(1:D))
  !dir$ simd
  r2(1:D) = r2(1:D) + A(:,4) * (x2(1:D) - x2(2:D+1))
  !dir$ simd
  r2(2:D+1) = r2(2:D+1) + A(:,4) * (x2(2:D+1) - x2(1:D))
  !dir$ simd
  r3(1:D) = r3(1:D) + A(:,5) * (x3(1:D) - x3(2:D+1))
  !dir$ simd
  r3(2:D+1) = r3(2:D+1) + A(:,5) * (x3(2:D+1) - x3(1:D))

  /top horizontal contributions
  !dir$ simd
  r1(2:D+1) = r1(2:D+1) + A(:,6) * (x1(2:D+1) - x2(2:D+1))
  !dir$ simd
```

Chapter 7. Optimization of sam(oa)² for Efficiency

We add a `!dir$ simd` directive before each FORTRAN loop. In Table 7.1, one can see that this implementation does not give any performance gain. The `apply3D()` function does not terminate with segmentation fault, but it might produce wrong results, because we have tried to vectorize loops, which does not access the variables in aligned manner. For example, the vertical and top horizontal contributions in the `apply3D()` function access `r1`, `r2`, `r3` or `x1`, `x2`, `x3` arrays from the second element.

### 7.1.4 Optimized Version 3.2: Improved SIMD Instructions

In version 3.2, we take away all the SIMD instructions for arrays that do not guarantee aligned access, which results in the implementation shown in Listing 7.7.

#### Listing 7.7: Reduction of the Number of Used SIMD Directives

In the `apply3D()` function, we insert the `!dir$ simd` directive only before loops for which we are sure that they can be vectorized and have aligned access.

```fortran
subroutine apply3D(x1, x2, x3, r1, r2, r3, A)
    real, contiguous, intent(in) :: x1(:), x2(:), x3(:)
    real, contiguous, intent(inout) :: r1(:), r2(:), r3(:)
    real, contiguous, intent(in) :: A(:,:)
    ! dir$ assume_aligned x1:64, x2:64, x3:64, r1:64, r2:64, r3:64
    ! bottom horizontal contributions
    ! dir$ simd
    r1(1:D) = r1(1:D) + A(:,1) * (x1(1:D) - x2(1:D))
    ! dir$ simd
    r2(1:D) = r2(1:D) + A(:,1) * (x2(1:D) - x1(1:D))
    ! dir$ simd
    r3(1:D) = r3(1:D) + A(:,2) * (x3(1:D) - x2(1:D))
    ! dir$ simd
    r2(1:D) = r2(1:D) + A(:,2) * (x2(1:D) - x3(1:D))
    ! vertical contributions
    r1(1:D) = r1(1:D) + A(:,3) * (x1(1:D) - x1(2:D+1))
    r1(2:D+1) = r1(2:D+1) + A(:,3) * (x1(2:D+1) - x1(1:D))
    r2(1:D) = r2(1:D) + A(:,4) * (x2(1:D) - x2(2:D+1))
    r2(2:D+1) = r2(2:D+1) + A(:,4) * (x2(2:D+1) - x2(1:D))
    r3(1:D) = r3(1:D) + A(:,5) * (x3(1:D) - x3(2:D+1))
end subroutine
```
\[ r_3(2:D+1) = r_3(2:D+1) + A(:,5) \cdot (x_3(2:D+1) - x_3(1:D)) \]

/ top horizontal contributions
\[ r_1(2:D+1) = r_1(2:D+1) + A(:,6) \cdot (x_1(2:D+1) - x_2(2:D+1)) \]
\[ r_2(2:D+1) = r_2(2:D+1) + A(:,6) \cdot (x_2(2:D+1) - x_1(2:D+1)) \]
\[ r_3(2:D+1) = r_3(2:D+1) + A(:,7) \cdot (x_3(2:D+1) - x_2(2:D+1)) \]
\[ r_2(2:D+1) = r_2(2:D+1) + A(:,7) \cdot (x_2(2:D+1) - x_3(2:D+1)) \]

end subroutine

More information about the mentioned optimization strategies can be found in Section 3.3.

As a comparison between the different optimization versions, we prepare the `test_apply3D()` function, in which the arrays are initialized and the `apply3D()` is called within a loop consisting of 1.18 M iterations. The corresponding part of the code is defined as OpenMP parallel loop distributed equally between 118 threads using static scheduling. The number of vertical layers is set to its maximum value \( D=85 \). For the last test version 3.2 with reduced number of SIMD instructions, the tests are performed with \( D=85 \) and with \( D=80 \), which will give us more insights on how the performance changes, when the code is better vectorized in vertical direction. The measurement of the wall-clock time is started before the loop and stopped after it. The tests are performed on a SuperMIC node using only one Intel® Xeon Phi™ coprocessor card in native mode. In Table 7.1, the execution time for all the previously described versions plus the original one are shown.

<table>
<thead>
<tr>
<th>Program Version</th>
<th>Vertical Layers</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Version</td>
<td>85</td>
<td>544.35 seconds</td>
</tr>
<tr>
<td>Version 1: Contiguous Arrays</td>
<td>85</td>
<td>525.81 seconds</td>
</tr>
<tr>
<td>Version 2: Data Alignment</td>
<td>85</td>
<td>515.62 seconds</td>
</tr>
<tr>
<td>Version 3.1: SIMD Instructions</td>
<td>85</td>
<td>563.33 seconds</td>
</tr>
<tr>
<td>Version 3.2: Improved SIMD Instructions</td>
<td>85</td>
<td>525.97 seconds</td>
</tr>
<tr>
<td>Version 3.2: Improved SIMD Instructions</td>
<td>80</td>
<td>368.246 seconds</td>
</tr>
</tbody>
</table>

The results show that using contiguous arrays and data alignment is slightly beneficial for the execution of the `apply3D()` subroutine. The version 3.1 with included
SIMD directives for all loops performs slower than the original version. This suggests that vectorization of all the FORTRAN loops does not yield performance gains. The SIMD directives vectorization is aggressive, which often leads to incorrect results, when the directive is used inappropriate. Therefore, we implement a second SIMD version – version 3.2 – using carefully the SIMD directives. We add a `!dir$ simd` directive only for loops, which have guaranteed aligned access. The execution time for the second SIMD version is lower than the first one. However, it does not outperform version 2, using only data alignment without SIMD directives. We execute a second scenario with number of vertical layers dividable by 8. Using $D=80$ layers, we guarantee that the arrays in `apply3D()` subroutine vectorize better than before. The execution time improves by 150 seconds.

Aiming optimization of the sam(oa)$^2$ pressure solver, we will include the code transformations based on version 3.2 in the sam(oa)$^2$ framework. This is described in detail in Section 7.2.

### 7.2 Code Transformations of sam(oa)$^2$

As a next step of the optimization process, we introduce some of the code transformations from Section 7.1 to the sam(oa)$^2$ framework.

sam(oa)$^2$ framework supports Finite Element, Finite Volume and Discontinuous Galerkin discretizations without changes of the program core. This is possible by using an event-based interface, which calls the kernels dependent on the executed scenario [18]. In order to optimize the code, we have to go into more details about the software design and the initialization of the arrays, which we use in the pressure linear solver.

Based on the software design of sam(oa)$^2$, the arrays $x_1, x_2, x_3, r_1, r_2, r_3$ with size $(D+1)$, which we want to align in the `apply3D()` function, are initialized as two large arrays with size $3 \cdot (D+1)$ called $d$ and $v$. Figure 7.1 shows how these arrays are initialized. In the pressure solver, $d$ and $v$ are split into chunks and used as separate arrays.

The large vectors $d$ and $v$ are not copied into short vectors. Regardless of the arrays we use – short or large, we access the same memory allocations. This means that even if the large vectors are initialized as aligned, we cannot ensure that the second and the third chunk ($x_2, r_2$ and $x_3, r_3$, respectively) of the short vectors are aligned in the memory. If the array size $(D+1)$ is divisible by 8,
Chapter 7. Optimization of sam(oa)$^2$ for Efficiency

Large Arrays:
\[d, v\]
\[3(D+1)\]

Short Arrays:
\[r_1, x_1\]
\[(D+1)\]
\[r_2, x_2\]
\[(D+1)\]
\[r_3, x_3\]
\[(D+1)\]

Initialization Pressure Solver

sam(oa)$^2$ operates on the large arrays.

Figure 7.1: Initialization and Usage of Vectors $d$ and $v$ in sam(oa)$^2$. In sam(oa)$^2$ framework, the arrays $x_1, x_2, x_3, r_1, r_2, r_3$ are initialized as two large arrays with size $3(D+1)$ each. Using the large arrays, sam(oa)$^2$ executes complicated operations (like Discontinuous Galerkin method) in a consistent manner. Even if the arrays $v$ and $d$ are initialized as aligned arrays, the second and the third chunk (depicted in blue and green) will not be automatically aligned. This implies, that also in the pressure solver function `apply3D()` the short arrays $x_2, x_3$ and $r_1, r_2$ will not have aligned access.

we can align the starting address of the large vectors $d, v$, and then the boundaries of the short vectors $x_1, x_2, x_3, r_1, r_2, r_3$ will be automatically aligned. This is beneficial only if the number of vertical layers $(D+1)$ is multiple of 8, otherwise the data is not vectorized and thus the performance is not improved.

This problem can be solved in two different ways. The first option is to copy the data from the large arrays $d$ and $v$ into small ones. This action will increase the memory traffic. The Roofline model from Chapter 5 shows that the kernel of the pressure linear solver is already memory-bound, which means that any additional memory operation must be avoided. The second option is to use padding, which guarantees that each element of a new row lies on the desired address boundary by allocating an appropriate additional number of bits. In Figure 7.2, padding is used on the large vectors. We make sure that each chunk (depicted in red, blue and green color in Figure 7.2) of the large arrays has a specific memory boundary by increasing slightly the size of the chunks.
Chapter 7. Optimization of sam(oa)$^2$ for Efficiency

Figure 7.2: Padding of the Vectors $d$ and $v$ in sam(oa)$^2$. Padding can be used to align properly all the chunks of the $v$ and $d$ arrays. When we apply padding, we allocate more space for our arrays and ensure that each chunk lies on the desired address (for the MIC architecture this is 64 Bytes). In such a way, the short arrays $x_1$, $x_2$, $x_3$, $r_1$, $r_2$, $r_3$ from the apply3D() subroutine are also aligned and can be vectorized easier by the compiler. The main drawback of this implementation is that sam(oa)$^2$ framework has to be further modified to execute properly the padded large arrays.

The padding of the large arrays leads to padded and aligned short arrays, which improves the vectorization in the pressure solver kernel. However, if we change the large arrays $d$ and $v$ and introduce gaps between the elements, we have to make multiple modifications in the entire sam(oa)$^2$ core. Whenever sam(oa)$^2$ operates on the large arrays, we have to guarantee that the functions access the right memory locations for the operations. Thus, padding of the arrays $d$ and $v$ will not be implemented because it is beyond the scope of this thesis.

The implemented changes in the sam(oa)$^2$ framework are based on version 3.2 from Section 7.1. We define the large arrays $d$ and $v$ to be aligned in the memory as shown in Listing 7.8.
Listing 7.8: Arrays $d$ and $v$ Declared As Aligned. The large arrays $d$ and $v$ are declared and the $\text{!dir$ attributes align:64}$ is used to guarantee that the arrays are allocated in aligned fashion.

\begin{verbatim}
real :: d (3*(D+1))
real :: v (3*(D+1))
!dir$ attributes align:64 :: d
!dir$ attributes align:64 :: v
\end{verbatim}

In the $\text{apply3D}$ subroutine, we set the variables $x_1, x_2, x_3, r_1, r_2, r_3$ to be contiguous and add the FORTRAN directives $\text{!dir$ assume_aligned var:64}$ and $\text{!dir$ simd}$ as shown in Listing 7.7.

The optimized sam(oa)$^2$ version is built for symmetric execution mode on the Intel® MIC architecture. Experiments and performance analysis are explained in Section 7.3.

### 7.3 Optimized Pressure Kernel – Single Node Performance

In the current Section 7.3, we want to show how the changes of the code described in Chapter 7 affect the performance of the 2.5D porous media flow in sam(oa)$^2$. The tests are executed on one SuperMIC node using 118 MPI ranks on each Intel® Xeon Phi™ coprocessor, and 16 MPI ranks on the Ivy Bridge host. The scenario has 8 vertical layers, with 2 M elements and 8 Sierpinski sections per core.

For the optimization of the pressure solver we have used SIMD directives $\text{!dir$ simd}$, and we have tried to align the access to the small arrays $r_1, r_2, r_3, x_1, x_2, x_3$ in the $\text{apply3D}$ function. Using these directives, the compiler builds two versions of the code – a first one, in which the data is aligned and the SIMD instructions are allowed, and a second, in which the compiler ignores the directives. The decision, which execution to be used, is made at runtime, which leads to slower execution. It is also possible that by using these directives, we have forced the compiler to make changes on the code like vectorization, which is not beneficial for the performance and leads to inefficient execution.

Figure 7.3 shows the memory throughput of a vectorized and non-vectorized sam(oa)$^2$ execution. In Figure 7.3, one can see that at the beginning of the run, the memory throughput is very low. During the zero phase, the grid is initialized and refined, which explains the low values. Within the first and second phases, the memory throughput is still low because of the time-based load balancing algorithm, which needs 10-20 time steps to converge. After the second phase, the memory
Figure 7.3: Comparison of the Memory Throughput Between a Vectorized and a Non-Vectorized Execution. The memory throughput values from a simulation with vectorized (red) and non-vectorized code (blue) are given. At the beginning of the simulation, the memory throughput is very low, because the grid is initialized and refined, and not a lot of data is used for this steps. After the initialization phase, the memory throughput increases and saturates with 45.8 GB/s and 49.7 GB/s for the vectorized and non-vectorized version of the pressure solver, respectively (high number is good).

throughput saturates with 45.8% and 49.7% for the vectorized and non-vectorized version, respectively. However, the vectorized code performs worse than the original one. The difference between both executions is not very large – approximately 3.9 GB/s.

In Figure 7.4, the wall-clock time for each of the phases is displayed. The phase number zero gives the time spent in grid initialization and refinement. This part is usually very fast with low element and memory throughput. The next phase takes approximately 30% more time than the other phases, without considering the initialization phase. As mentioned before, the reason is the time-based load balancing algorithm, which needs some iterations to converge. After the second phase, the wall-clock time does not change significantly.

The non-vectorized version of the sam(oa)$^2$ framework runs slightly faster than the vectorized one. In some phases, the difference in memory throughput between
Chapter 7. Optimization of $\text{sam}(oa)^2$ for Efficiency

Figure 7.4: Comparison Between the Runtime of a Vectorized and a Non-Vectorization Version of $\text{sam}(oa)^2$. The initialization phase (phase zero) is executed very fast, while the next phase takes the longest time of the simulation, because there is some imbalance of the load per core. After that, the execution time stays stable. The non-optimized version of the pressure solver runs slightly faster than the optimized one. The reason for this might be forced inefficient vectorization (low number is good).

The initialization phase is executed very fast, while the next phase takes the longest time of the simulation, because there is some imbalance of the load per core. After that, the execution time stays stable. The non-optimized version of the pressure solver runs slightly faster than the optimized one. The reason for this might be forced inefficient vectorization (low number is good).

them is smaller, but the non-vectorized code tends to be approximately 60 seconds faster.

To understand better the behavior of both simulations, we can break down the wall-clock time into the following components – computation, synchronization, barrier, load balancing and (de)allocation. Figure 7.5 shows computation, synchronization, barrier and total time for the vectorized and non-vectorized executions. The load balancing and (de)allocation components have very low execution time. Therefore, they are not visualized in the figure.

Both runs do not differ a lot in the time spent for each component. The computation time is slightly higher for the vectorized version of the $\text{apply3D()}$ function, while the synchronization and barrier time have approximately the same values. This and the detailed information about the optimization in $\text{sam}(oa)^2$ from Section 7.1 and Section 7.2 suggest that even using the SIMD and align directives, the data is not fully aligned and the $\text{apply3D()}$ function of the pressure solver is
Figure 7.5: Comparison Between the Components of the Vectorized and Non-Vectorized Version of sam(oa)$^2$. The wall-clock time is broken down to computation time, synchronization time and time spend at barriers. The optimized code needs a little bit more time for computation, but in all the other components there is almost no difference (low number is better).

not completely vectorized. The compiler builds correct code, but without any performance gain. The vectorization of the code depends very strong on the number of vertical layers. If the number of layers $(D+1)$ is divisible by 8, the 2.5D porous media flow scenario scales much better and takes advantage of the data alignment directives and the SIMD vector instructions.
Chapter 8

Conclusions

In this thesis, we successfully compiled and executed the 2.5D porous media flow scenario of the sam(oa)$^2$ framework on the Intel® Xeon Phi$^\text{TM}$ architecture. The building process was formulated in a way that the platform independence of the sam(oa)$^2$ framework was preserved. The Roofline model was applied to the pressure linear solver of the 2.5D porous media flow to gain knowledge about realistic performance expectations and to identify the potential bottlenecks. The Roofline model’s 2D graph showed that the performance is clearly limited by memory bandwidth. Therefore, we concentrated on the memory throughput of sam(oa)$^2$ as a characteristic performance metric. The results from the simulations executed on the SuperMIC cluster showed that the 2.5D flow scenario achieved only 35% of the memory throughput of the STREAM benchmark. The sam(oa)$^2$ framework performed as good as possible, but the Intel® Xeon Phi$^\text{TM}$ nodes have very high memory bandwidth, which is hard to exploit even for more computational demanding algorithms. However, the sam(oa)$^2$ framework did not scale well due to load imbalance.

All the tests executed on the Intel® Xeon Phi$^\text{TM}$ architecture were using symmetric mode, which allows distributed and shared memory parallelism and utilizes the whole SuperMIC node. The mode was applied without expensive code changes. Based on the results from Section 6.2, the symmetric mode halved the time-to-solution with respect to the host only and coprocessors only executions. It gave the best possible performance without modifications of the sam(oa)$^2$ framework. However, the symmetric mode uses a heterogeneous hardware, which is a source of load balancing problems. We compared the two already implemented load balancing algorithms in sam(oa)$^2$– time-based and cell-based. Both algorithms are not designed to run on a heterogeneous hardware, but the time-based load
balancing collects data and analysis it, so that it is able to tackle the issues caused by the heterogeneity of the Intel® Xeon Phi™ system. This characteristic has an impact on the performance, which decreased in the first time steps, because the time spent at barriers dominated in the execution. After the time-based load balancing algorithm converged, the barrier time decreased.

However, if more than one SuperMIC node was used and the simulation domain grew proportionally to the number of nodes (weak scaling) the imbalance of the initial time steps eventually led to early termination of the execution. This happened, because the nodes got overloaded and ran out of memory. To obtain better results, we have to improve the time-based load balancing algorithm or implement a new one, which deals with the heterogeneity. The load location (host or processor) has to be accounted and thus, more appropriately distributed between the processes. This task should be further investigated and implemented within the sam(oa)² framework.

Besides the pure MPI parallelization, we tested also the hybrid MPI+OpenMP parallelization, which was expected to decrease the memory per core and reduce the execution time exploiting simultaneously distributed and shared memory parallelism. However, the memory throughput of the hybrid execution was much lower than the one on the pure MPI execution. The reason for this is the extremely dominating synchronization time, which implies high global communication and imbalanced load between the processes, on which further investigation needs to be done.

This work can be extended by developing a new load balancing algorithm or improving the time-based load balancing in such a way that the heterogeneity issues are captured and successfully eliminated. Another aspect of performance improvement that can be worked upon is to find better optimization and vectorization strategies, which can be applied without restricting the sam(oa)² framework.
Bibliography


