HiPowerEd

DIGITAL EMPOWERING TROUGH
HPC EDUCATION

HiPowerED consortium

March 8, 2023

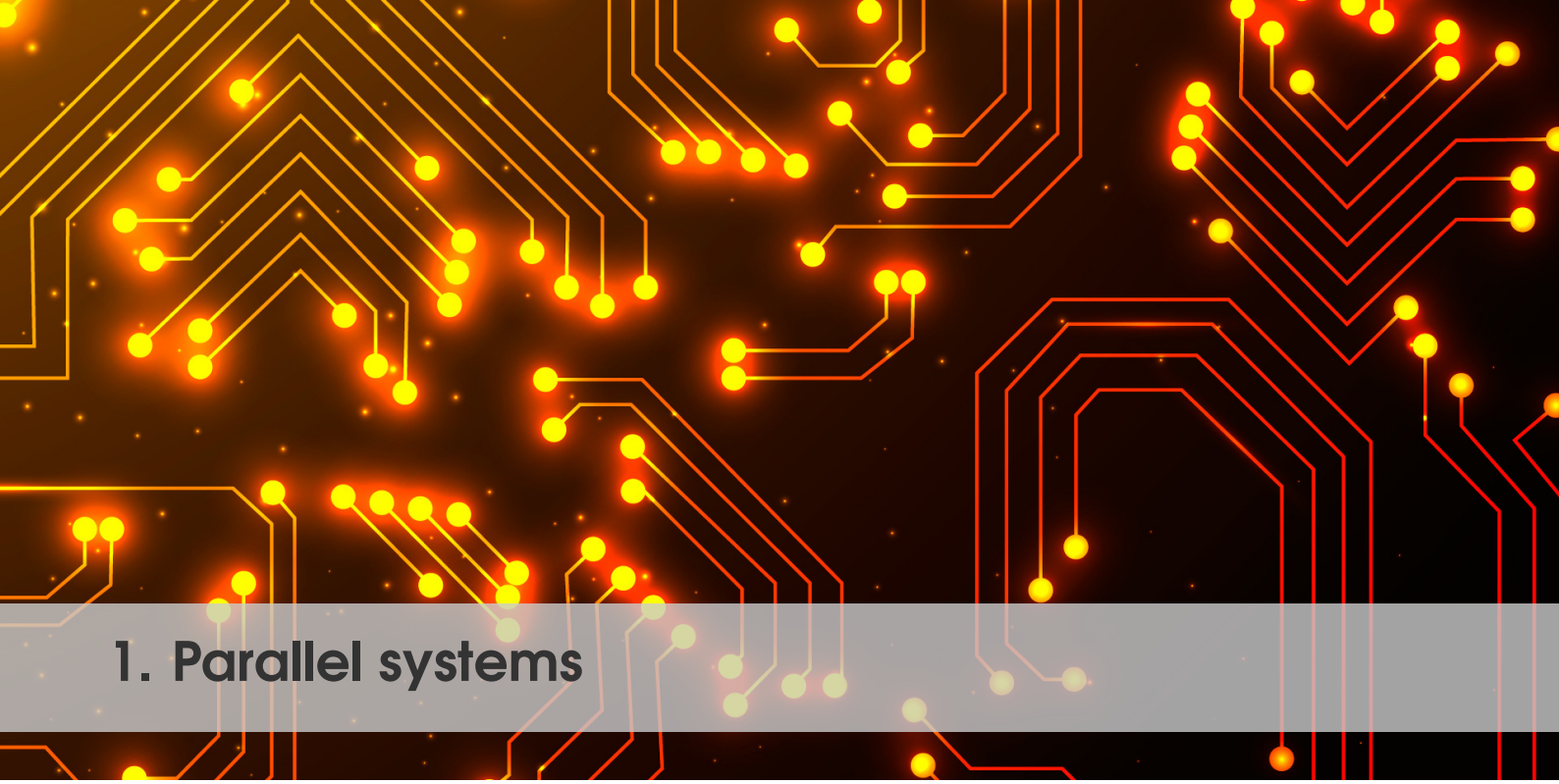
Technical University of Denmark
Technical University of Munich
University College Algebra
University of Rijeka
University of Trieste

March 8, 2023

Contents

Part I

Parallel computing



1. Parallel systems

1.1 Introduction

University of Rijeka

Parallelism is nowadays evident in a majority of the devices that surround us; computers, phones and a wide range of IoT devices utilize some type of parallelism, i.e. they execute certain tasks concurrently. On a hardware level, processor microarchitectures are inherently parallel. Processors are built to be able to execute multiple instructions at a time and typically have multiple cores, each of which can execute its own set of instructions. Graphic processors are an extension of this principle and a prime example of massive parallelism at a hardware level. Hardware advancements, however, without proper software support are meaningless. Finding solutions for comprehensive tasks, problems and domains is typically not possible on either traditional or modern computer architectures if they utilize sequential code. The serial computing concept (computing on a single processor/using a single process) is a limiting factor since complex problems require an immense amount of time. Consequently, parallel computing was born. In essence, parallel computing incorporates hardware components and software into a system that allows seamless simultaneous execution of calculations/code.

The complexity of engineering problems has over the past decades grown exponentially. Previously conducted experimental tasks are commonly replaced with extensive and comprehensive simulations. This transition is not exclusive to engineering; biochemists and pharmaceutical companies utilize supercomputing resources to simulate protein folding, which is essential in the development of new drugs. Machine learning and AI rely on extensive computational resources to formulate models which provide insight into complex interactions and allow event predictions. In the field of fluid mechanics, mechanical engineers simulate interactions in vast domains where said physical domains are described through high-resolution networks of nodes and elements. Pressed by short turnover times, parallel computing has become invaluable to scientists and engineers.

1.2 Computer architecture

University of Rijeka

Historically, computers can be divided into two main groups: fixed program computers and stored program computers. The fundamental advantage of a stored computer is the ability to execute different tasks i.e. they can be programmed.

1.2.1 Von Neumann architecture

Modern computers are an extension of the stored-program concept introduced by von Neumann, often referred to as Princeton (or von Neumann) architecture [Weinzierl, 2021]. According to von Neumann, a computer should contain several key components:

- an arithmetic/logic unit (ALU).
- a control unit (CU).
- a memory unit (MU).
- input/output (I/O) devices.

ALU is designed to execute calculations. It incorporates registers, a form of local memory. CU manages devices and signaling, and directs instruction execution and data flow. Similarly, it has an instruction register. These two components are typically combined and constitute a central processing unit (CPU). The main memory contains all the data and instructions. Input and output devices allow the input of data (or new instructions) and output of the information to the user. Finally, buses are the main information interchange channels that ensure information transfer between different components of a computer. A schematic overview of a von Neumann computer is given in figure 1.1.

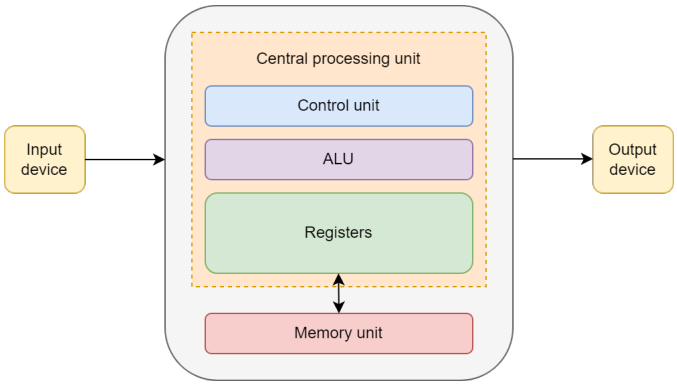


Figure 1.1 A von Neumann architecture.

Von Neumann architecture has its limitations which are mainly related to data transfer. In essence, the throughput limitations of system buses (the rate at which the data is fed to the CPU) force the CPU to idle while waiting for data. It is theorized that the only method to combat this is a radical new design. Nevertheless, different methods are being employed to mitigate this bottleneck such as caching (L1, L2, L3 cache), prefetching, speculative execution, multithreading, etc.

1.2.2 Harvard architecture

Harvard architecture can be interpreted as a variant of the von Neumann design where instruction and data storage are physically separated i.e. they use separate instruction and data caches as well as memory buses [Smith, 1988]. This is significant as it allows simultaneous access to the instructions and data thus partially overcoming the von Neumann bottleneck. A schematic overview of Harvard architecture is given in figure 1.2.

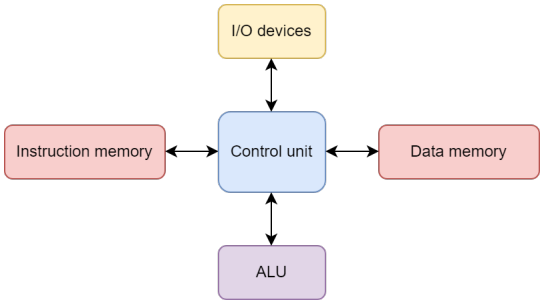


Figure 1.2 Harvard architecture.

A modified Harvard architecture is an extension of the proposed concept with relaxed stipulations regarding memory separation: a common address space is used, however, caches are separated. This architecture displaced the original concept and has mostly replaced it. Modern CPUs also employ this approach hence a CPU can be classified as modified Harvard architecture. However, a computer as a whole employs the von Neumann architecture (no separation in memory). Modified Harvard architecture is typically found in microcontrollers, digital signal processors, etc.

1.3 Moore's law

University of Rijeka

Moore's Law is an often-cited remark given in 1965 by the CEO of Intel, Gordon Moore, which states that the number of components (i.e. transistors, resistors, diodes, or capacitors) in integrated circuits doubles every year. This statement has been revised in 1975 and stipulates that the number doubles every two years [Dongarra and van der Steen, 2012]. This observation has held since and is thus often considered a "law".

The observation was based on then-current empirical evidence and is commonly linked to both the transistor count and the overall increase in performance. Figure 1.3 depicts the increase in the transistor count since the 70s.

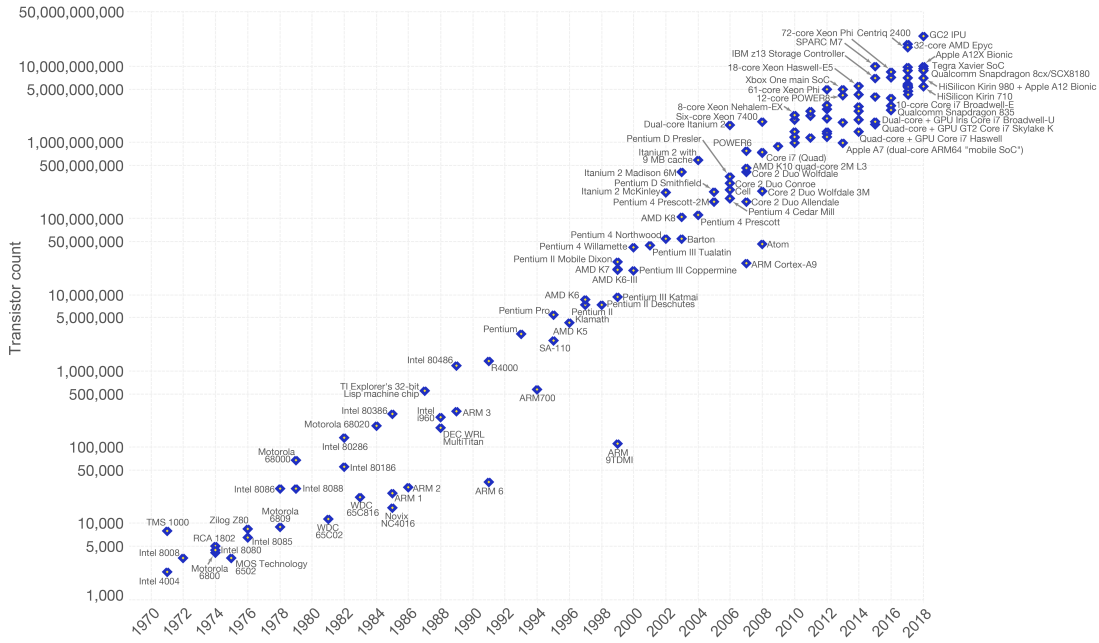


Figure 1.3 Increase in device transistor count since the early 70s [Roser et al., 2022].

Moore's Law has held for almost 50 years, although it somewhat slowed down as of 2016/2017 with estimates that say the period has increased from two years to approximately two and a half years. Recent calculations suggest it takes ≈ 2.1 years to double the transistor counts. In other words, certain computer minimization technologies are reaching the physical limits of what is possible, hence in order to keep the continuation of the increase in computing power, the development of new technologies and approaches is of great importance. Figure 1.4 depicts the yearly performance improvement on the TOP500 supercomputer list.

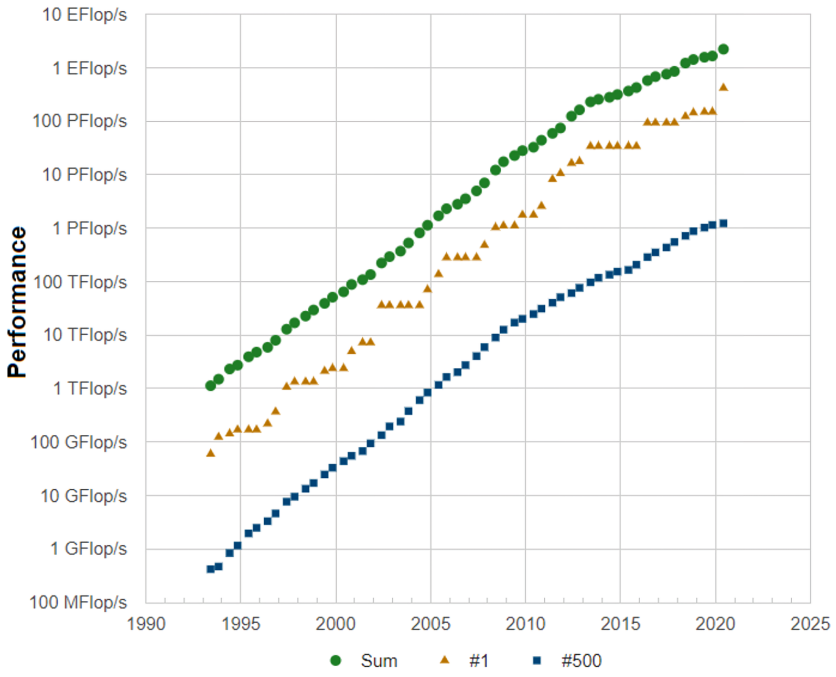


Figure 1.4 Supercomputer performance development since the early 90s [TOP500.org, 2022].

1.4 Parallel computer architectures

University of Rijeka

The overall structure of the computer i.e. the architecture of the computer, to a greater extent, determines the feasibility of a performance uplift (i.e. acceleration) above the serial/sequential performance. Programmer's input i.e. level of the code parallelism also greatly affects the speed of the execution. Another important factor is the ability of a program compiler to generate efficient code for a given computer platform. In many cases, it is difficult to assess and discern the impact hardware and software have on the computational speed (performance of the program).

Properties of a parallel computer and hardware specifics are typically expressed through architectural classification models which group similar performing computers based on performance and common features i.e. architecture. The most comprehensive classification methodology covering a wide range of computers from personal to vector computers and parallel computers with high performance is Flynn's taxonomy (1966) [Gebali, 2011]. The classification is based on four English letters:

- **S** single.
- **I** instruction.
- **M** multiple.
- **D** data.

These four letters are used to define four main and distinct types of computer architectures [Gebali, 2011]:

- **SISD** single instruction single data. Single processor machines.
- **SIMD** single instruction multiple data. Multiple processors. All processors execute the same instruction on different data.
- **MISD** multiple instruction single data. Systolic arrays. Uncommon.
- **MIMD** multiple instruction multiple data. Multicore processors and multithreaded multiprocessors. Each processor is running its instructions on its local data.

In addition to the previous classifications, parallel computers can be further divided into two principal groups based on memory organization [Gebali, 2011]:

- **multiprocessors** computers with shared memory.
- **multicomputers** computers with distributed memory.

1.4.1 SISD

This group represents conventional serial computers that consist of a single processor connected to the memory. The processor executes a program that specifies a series of read/write operations on memory. Such a computer is often called a von Neumann computer [Weinzierl, 2021] and the processor a scalar processor.

Home PCs from the early 21st century were mostly SISD while most of the software commonly in use today on home PCs still employs the SISD principle. By connecting multiple SISD computers through a computer network, an architecture known as

multiple SISD computer (multicomputer) can be derived, in which each processor executes commands independently of other processors. An example of such a parallel computer is the Beowulf cluster, classified as a distributed memory MIMD (DM-MIMD) machine [Becker et al., 1995].

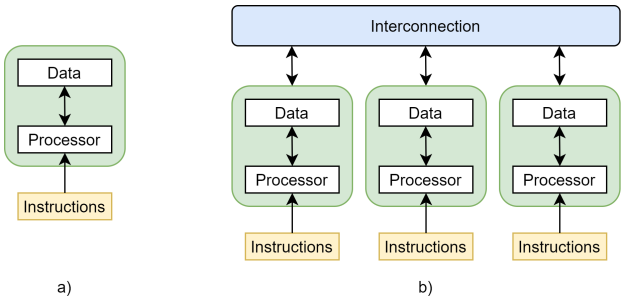


Figure 1.5 a) SISD computer, b) multicomputer.

1.4.2 SIMD

Single Instruction Multiple Data (SIMD) computers are a type of parallel computer. SI means all processing units execute the same command (instruction) at any time while MD refers to the fact that each processor can operate on a different data element. Instructions are conducted simultaneously [Gebali, 2011]. SIMD classification includes three types of computers that are very different in terms of architecture:

- parallel computers with a large number of processors (2^{10} to 2^{14}) which simultaneously execute the same commands on different data i.e. processor arrays. This type of computer is no longer relevant but is sometimes used for special purposes.
- vector computers that operate on vectors of similar data. They have a distinct processor structure that allows the processors to execute commands in quasi-parallel mode only when working with vectors and not scalars.
- computers based on GPGPU processors i.e. general purpose graphics processors [Nielsen, 2016].

SIMD computers can also be classified into two sub-classes based on memory organization:

- SIMD computers with shared memory.
- SIMD computers with distributed memory.

SIMD computers with shared memory, vector computers

This subclass of SIMD computers is essentially equivalent to a single-processor vector computer. A vector computer is a machine that performs arithmetic operations particularly efficiently on vectors and is hence especially important in scientific calculations where calculations with matrices and vectors are particularly common. Vector computers are several times faster when running operations on vectors rather than on scalars [Siegel, 1979, Gebali, 2011].

The key distinctiveness of a vector computer is its arithmetic unit, the so-called arithmetic tube, i.e. pipeline, which performs arithmetic operations on vector elements

consecutively, thus increasing the computational efficiency. The pipeline is similar to a production line in a factory; different processing sequences are performed on different parts of the product on the production line. For example, when summarizing two vectors x and y and using the floating-point approach, the operation $s = x + y$ can be defined with steps (a) to (f):

(a) the exponents of two floating-point scalars (which are elements of a vector) are compared to determine the smaller of the two exponents.
(b) the decimal part of a number with the smaller exponent is modified so that the exponents for both numbers are equal.
(c) decimal parts are added together.
(d) the summation result is normalized.
(e) validity control of the performed floating point operation.
(f) rounding of the result.

The process of adding two vectors, registers and arithmetic tubes can be seen in figure 1.6.

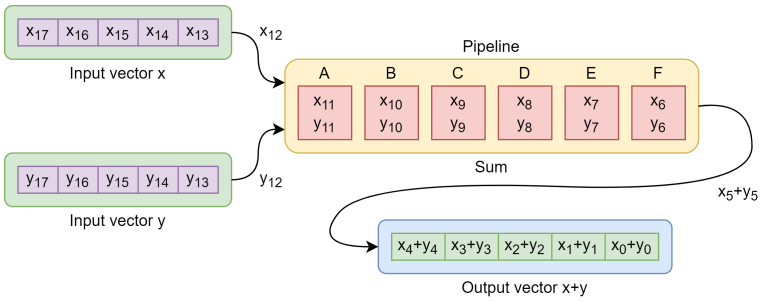


Figure 1.6 Vector addition on a vector computer.

Manufacturers of vector computers that were at the time leaders in the field of computing are CRAY, NEC, Hitachi, Fujitsu, etc.

SIMD computers with distributed memory

SIMD computers with distributed memory are sometimes referred to as processor-array machines or processor arrays. Each processor in an array executes the same command but on different data, with no need for mutual synchronization of the processors. Instructions that need to be executed by individual processors are regulated and issued by a central processor. Typically, processor arrays utilize a so-called front-end processor attached to the central processor which is used for I/O operations or offloading/calculations if the array or the central processors are unable to do so. The interconnection network used in this type of machine is always a 2D grid [Dongarra and van der Steen, 2012]. The main parts of a DM-SIMD machine are, therefore:

- control processor.
- processor array consisting of many processors (1024 or more) each with its memory.
- front-end processor.

With computers of this architecture, it is possible to turn off some of the processors in the array, under certain logical conditions. In that case, excluded processors are on hold, which consequently reduces the overall system performance. Another unfavorable situation is when the processor needs the data that is in the memory of

another processor. This data transfer impedes performance (long waiting times). This is especially problematic if the situation occurs simultaneously on multiple processors or even on all processors.

Therefore, in order to take the advantage of the positive aspects of a processor array, they are employed for specific tasks e.g. signal processing (digital signal, radar signal, etc.), image processing, Monte Carlo simulations, etc. It is evident that the presented use cases do not require or require minimal communication between processors in the array. An example of a distributed memory SIMD computer is given in figure 1.7.

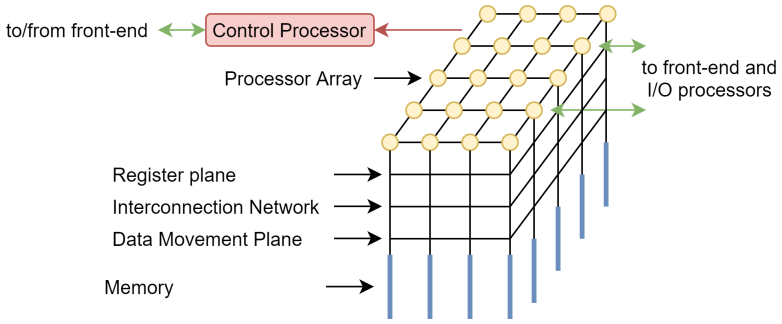


Figure 1.7 SIMD computer with distributed memory.

GPGPU processors

The majority of the high-performing computers today, both desktop and supercomputers, rely on CPU-GPU interdependence. GPGPU use allows for significant speedups in cases where so-called data-parallel models are to be evaluated. These include physics simulations, encryption/decryption, scientific computing, AI use, etc. [Trobec et al., 2020]. This advantage is, however, lost, in cases where considerable communication is needed i.e. communication between different threads.

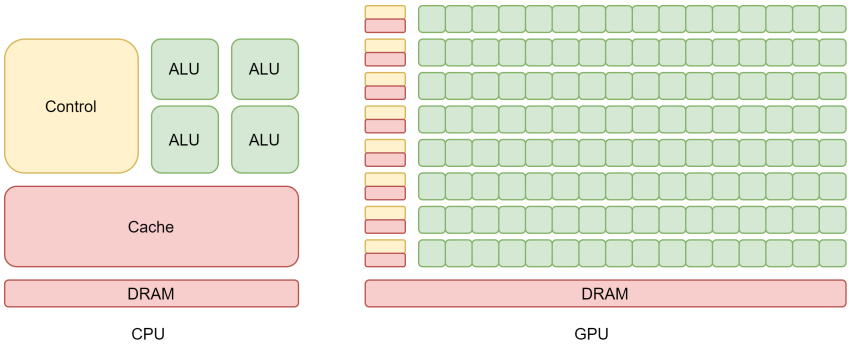


Figure 1.8 Schematic comparison of CPU and GPU architectures.

Generally speaking, each processor, whether in a CPU or a GPU, has its cache and access to shared DRAM. GPUs utilize many Arithmetic Logic Units (ALU) but lack the shared cache, which limits the overall communication between parallel threads.

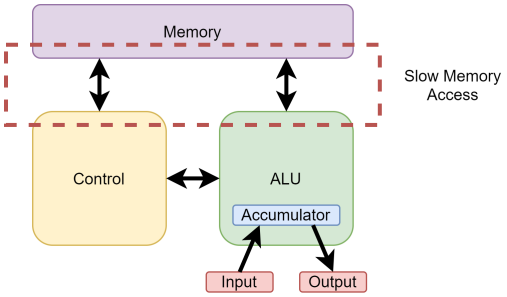


Figure 1.9 CPU bottleneck: communication pathway between memory and ALU.

On the other hand, CPUs, or rather their ALUs, have comparatively slow memory access i.e. this represents a bottleneck. Consequently, hybrid designs are usually employed i.e. GPGPU accelerated CPUs/nodes to take the advantage of both technologies.

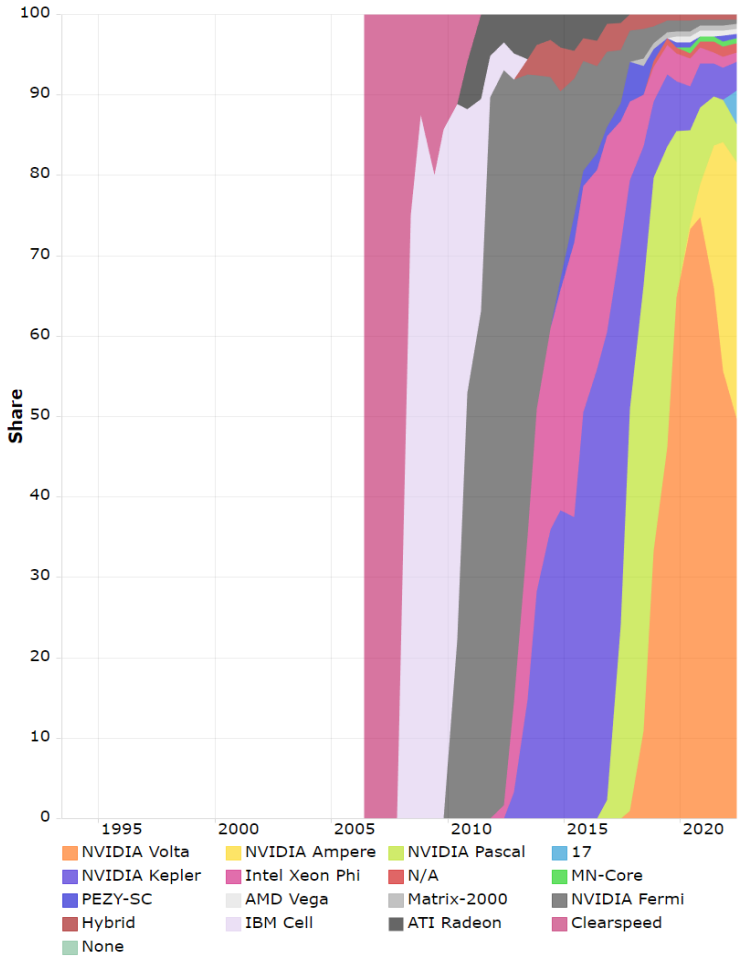


Figure 1.10 Use of accelerators in supercomputers. GPGPUs are increasingly common, with NVIDIA GPUs dominating the market [TOP500.org, 2022].

1.4.3 MISD

Multiple Instruction Single Data (MISD) machines are generally uncommon. It is argued that e.g. neural networks can be interpreted as the MISD concept representatives

[Gebali, 2011]. Since the fundamental requirement is the ability to conduct multiple operations on the same data, pipeline machines can also be seen as MISD machines. Another common example is systolic arrays. An overview of an MISD architecture is given in figure 1.11.

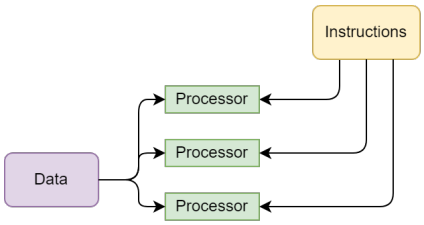


Figure 1.11 MISD architecture schematic.

1.4.4 MIMD

Multiple Instruction Multiple Data (MIMD) parallel computers are currently the most common. They are characterized by the ability to synchronously or asynchronously execute different instructions on different data. Modern multi-core computers belong to this category, although they can also include elements that due to their architecture might better correspond to some other category [Gebali, 2011]. MIMD computers will be analyzed according to respective memory arrangement concepts i.e. MIMD computers with shared memory (SM-MIMD) and MIMD computers with distributed memory (DM-MIMD) will be discussed.

MIMD computers with shared memory

Processors in shared memory MIMD systems (SM-MIMD) can concurrently perform different tasks and access the same address space of a common (shared) memory through an interconnection network. Memory coherence is typically managed by the operating system/software (cache coherency protocols). Hardware-based protocols do exist and usually offer faster mechanisms for maintaining memory consistency, however, they introduce hardware complexity and are thus not as common [Gebali, 2011]. The number of processors in an SM-MIMD system is rather small, typically less than 32. UMA (Uniform Memory Access) multiprocessors commonly referred to as SMP (Symmetric Multiprocessors) are viewed as SM-MIMD machines due to the centralized nature of the memory. The architecture of an SM-MIMD computer is given in figure 1.12.

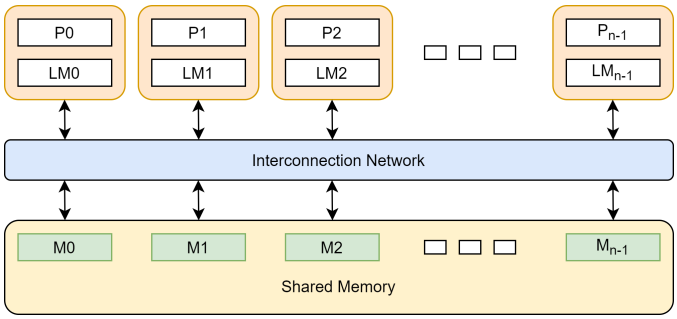


Figure 1.12 SM-MIMD system.

In addition to standard multiprocessor computers, special types of multiple vector

computers can also be considered SM-MIMD machines. Previously mentioned vector computers with a single vector processor can only be considered a special case of a more general MIMD classification as there are also vector computers with multiple vector processors. Multiprocessor vector computers use a crossbar network topology given that the maximum number of processors in such systems is relatively small and a low-performance network topology would be unsuitable for fast vector processors [Dongarra and van der Steen, 2012].

In today's high-performance systems, an architecture based on an exclusively shared memory concept is rare, with distributed memory systems being a more suitable alternative. This is mostly due to the limited scalability of shared memory systems. Memory access and associated bus contention are the main problems associated with SM-MIMD systems. They can be partially alleviated by including local caches. Commonly, crossbar or multi-stage crossbar networks are employed to ensure throughput hence to minimize the complexity, a lower number of CPUs is used.

ccNUMA (Cache Coherent Non-Uniform Memory Access) architectures are commonly considered SM-MIMD computers. This is mainly due to the fact that these computers, although they have physically distributed memory, from a programmer's standpoint, utilize a shared memory concept, which exists at a logical level (software-based) [Dongarra and van der Steen, 2012].

MIMD computers with distributed memory

MIMD parallel computers with distributed memory (DM-MIMD) are currently the most prevalent. They are often referred to as multicomputers. Unlike MIMD computers with shared memory (SM-MIMD) where the distribution of data is completely transparent to the user and accessible to all processors, users on a DM-MIMD system must explicitly distribute the data to each processor and explicitly regulate the data exchange between processors. Data access is accomplished through a network [Gebali, 2011, Dongarra and van der Steen, 2012]. Such requirements from the user/programmer and the overall programming complexity are the main reasons why this computer architecture was not widely accepted despite being available. The current resurgence and rapid development of DM-MIMD parallel computers can be attributed to:

- availability of mass-produced decently performing cheap processors (e.g. Intel).
- lack of technological advancements that have hampered further development of high-performance processors (e.g. RISC).
- development of standards for communication software, which includes MPI (Message Passing Interface) and older PVM (Parallel Virtual Machine) message-passing standards.
- development of highly-scalable interconnection networks.

By transferring a segment of the parallelization load from the hardware level to the software/communication level, as is the case with DM-MIMD parallel computers, despite drawbacks, certain benefits can be gained, amongst which is the most important the not-so-theoretical ability to outperform any other architecture. Furthermore, unlike shared memory systems, the bandwidth scales well with the number of processors. The downsides of the DM-MIMD architecture include:

- communication between processors is slower than on SM-MIMD machines, hence the synchronization overhead is of an order of magnitude higher.

- a large disparity in the access times for data stored locally and data stored in the memory of another processor forces programmers to carefully develop and structure their code in order to minimize access times to the data that is not stored locally. Consequently, codes are usually more complex.
- inadequate load balancing across distributed processors can hinder performance.

It is obvious that DM-MIMD parallel machines rely on high-performing interconnection networks, their topology and bandwidth, which are the main limiting factors in the overall performance efficiency of a DM-MIMD system [Dongarra and van der Steen, 2012]. Figure 1.13 depicts a DM-MIMD machine.

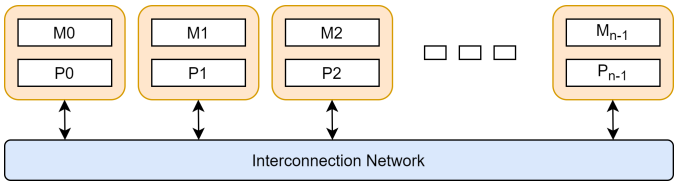


Figure 1.13 DM-MIMD system.

1.4.5 Cluster, grid and other concepts

Massively Parallel Processing (MPP) and Symmetric Multi-Processing (SMP) computers were the dominant high-performance computers during the second half of the 90s and at the beginning of this century. MPP computers were comprised of a larger volume of RISC processors. SMPs, although often classified as slightly slower performing computers, had higher prevalence and were still sufficiently performant to be used in multi-processor systems where processors have been typically connected through crossbar interconnection networks and shared common memory space. Reported single-processor machine designs include both single-processor and vector computers. SIMD computers were somewhat relevant during the 90s, however, they have mostly disappeared since.

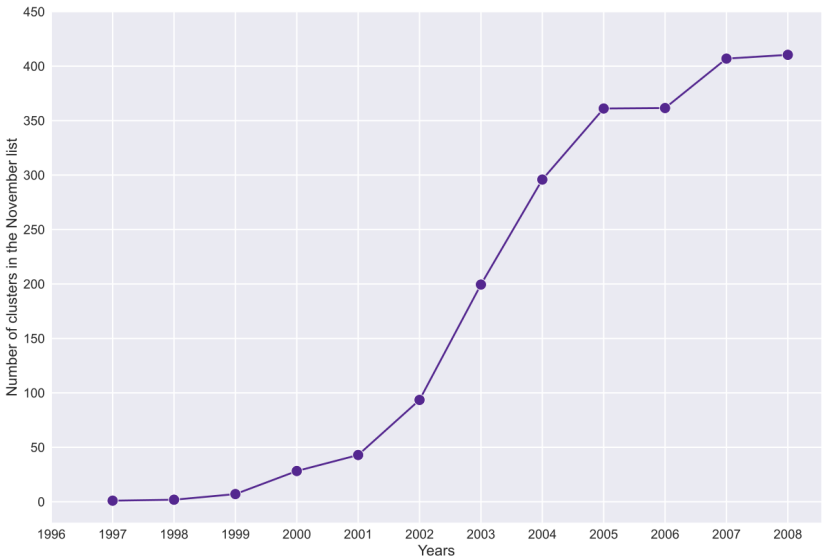


Figure 1.14 A surge in the number of clusters occurred around the 2002-2004 period [TOP500.org, 2022].

At the beginning of this century, the rise of the clusters began, with an inflection point being reached in late 2003 when the architecture began to dominate as shown in figure 1.14. Clusters and MPPs today account for the majority of the most powerful supercomputers, as evidenced by figure 1.15, which depicts the representation of individual computer architectures among the 500 most powerful supercomputers in the last thirty years. The TOP500 list tracks the most powerful computers in the world according to the standard LINPACK test.

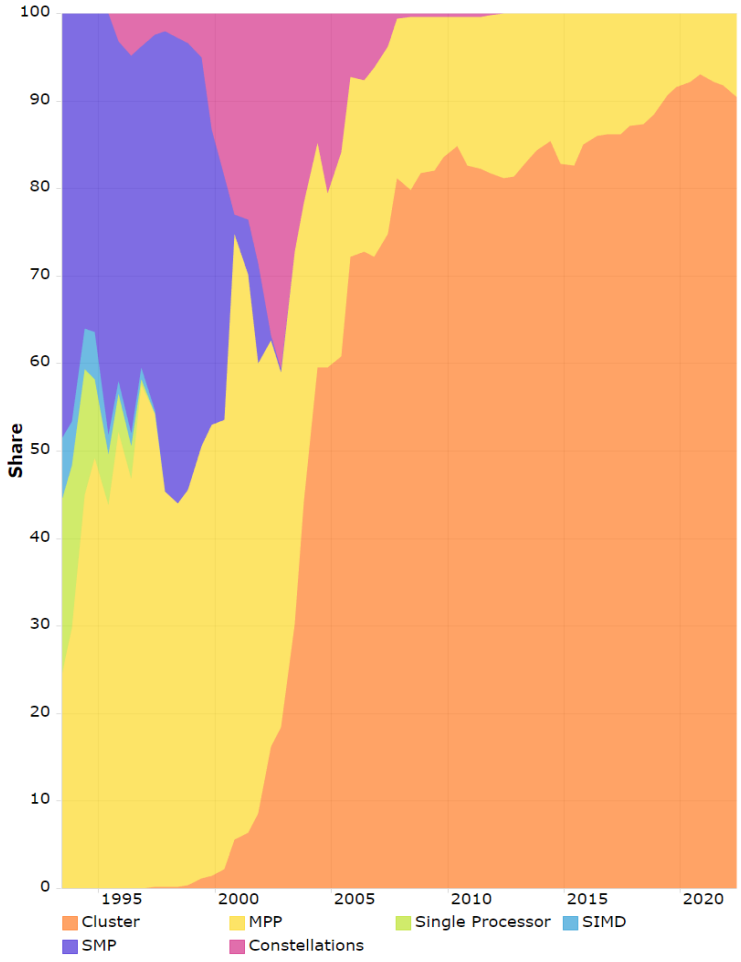


Figure 1.15 Distribution of major computer architectures in the TOP500 rankings from 1993 to 2022 [TOP500.org, 2022].

The progressive growth of clusters and constellations throughout the twenty-first century can be attributed to the overwhelming availability of cheap AMD and Intel processors. Intel processors are nowadays prevalent in supercomputers, with IBM and IBM-based custom designs still being utilized in some of the best-performing supercomputers. RISC-based (Sunway, CN) and ARM (Fugaku, JP) architectures are also present. AMD has only recently managed to revitalize its supercomputer business (Frontier, US). Since the early 2010s, CPUs have been increasingly coupled with accelerators. Nowadays, all the major systems utilize, typically Nvidia or AMD-provided, accelerators (GPGPUs). Figure 1.16 shows the distribution of different chip architectures from 1993 to 2022. The dominance of the x86-64 architecture and the downfall of the vector machines and RISC processors is evident.

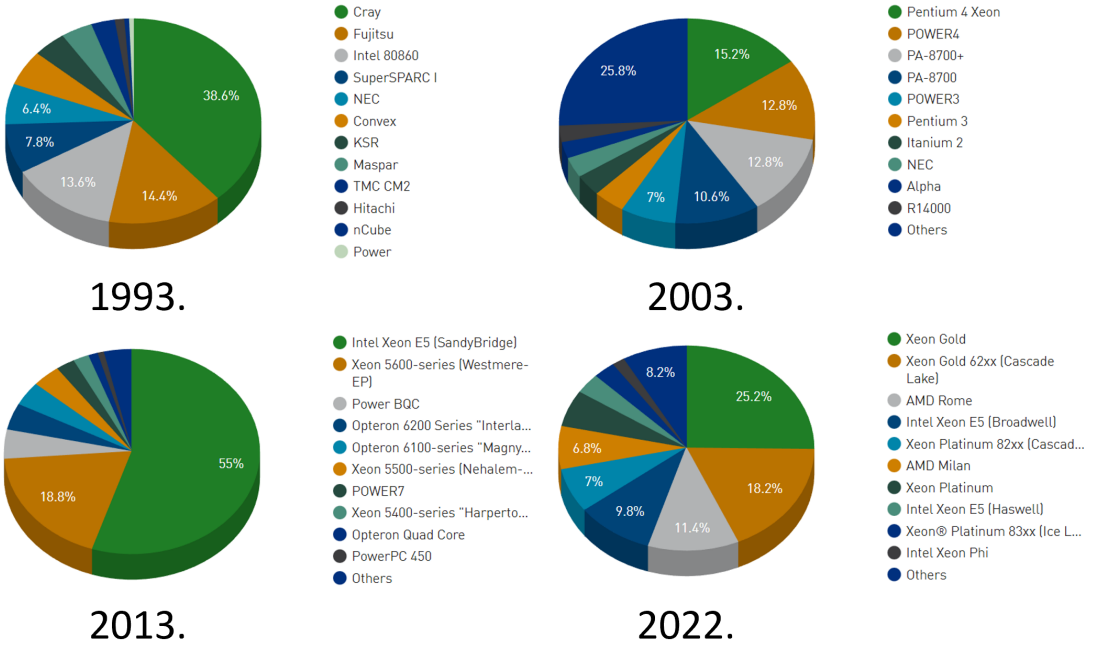


Figure 1.16 Distribution of chip architectures (technologies) in TOP500 supercomputers [TOP500.org, 2022].

The current HPC performance leader is the Frontier supercomputer located in Oak Ridge National Laboratory, United States. It is comprised of 74 HPE Cray EX cabinets that contain AMD EPYC processors and AMD Instinct accelerators. The system has more than 8.7 million CPU cores and 37.000 GPUs. At its peak, it is more than three times faster than the #2 supercomputer. Frontier is the first system that broke the exascale barrier with a performance of 1.1 EFlop/s [TOP500.org, 2022].

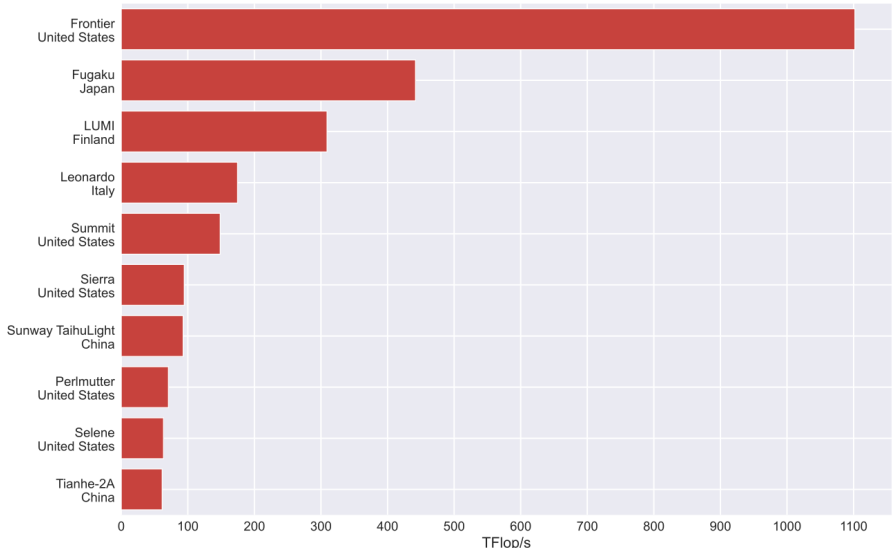


Figure 1.17 Top performing supercomputers as of November 2022 [TOP500.org, 2022].

Cluster

The concept of cluster architecture experienced a strong surge a decade after the emergence of the first cluster named Beowulf (NASA, 1994). Beowulf clusters in general are simple computer clusters composed of common PCs connected by a local area network. They are classified as multicomputers as they were originally built by connecting multiple SISD computers [Dongarra and van der Steen, 2012]. Beowulf nowadays represents a technology (methodology) of clustering computers to form a parallel supercomputer. There are no software requirements that would define a cluster as a Beowulf, although they typically do use Unix-like operating systems and rely on MPI (Message Passing Interface). The development of cluster technology incentivized large computer manufacturers to build their own clusters, hence the Beowulf concept became mostly obsolete and its use is limited to scientific computing. The structure of a typical cluster is given in figure 1.18.

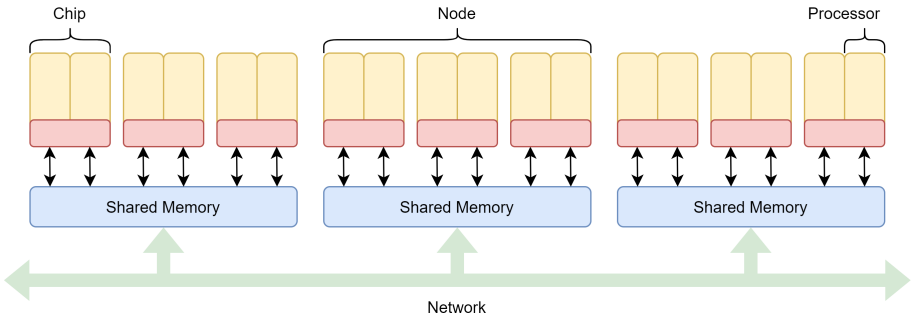


Figure 1.18 Diagram of a typical modern cluster. Compute nodes are comprised of multiple processors connected with a fast interconnect (e.g. crossbar). Multiple nodes are connected with an interconnection network such as a fat tree, hypercube or torus.

The displayed schematic defines an architecture in which SMP (Symmetric Multiprocessing) nodes are connected in a cluster through an interconnection network. Processors inside the SMP nodes (intranodal) are usually connected with a fast crossbar interconnection which allows a smaller number of processors to directly communicate with each other. Internodal connection is usually much slower, however, it is able to connect a large number of nodes into a functional system. System manufacturers typically use proprietary interconnections (Quadrics, Slingshot, etc.) or one of the alternatives such as 1GEthernet, 10GEthernet, Infiniband, etc. to connect the nodes.

The core of an SMP-based cluster are massive compute nodes. Compute nodes consist of one or more processors with shared memory. The number of processors per compute node to a larger extent depends on the purpose of the system and target use (i.e. application use), as well as on the manufacturer and the proposed design of the system. Nodes are based on the shared memory principle (SM-MIMD) with the whole system viewed as a parallel computer with distributed memory (DM-MIMD) and called a cluster (in the narrow sense).

Constellation

Term constellation (in the narrow sense) refers to a specific type of cluster computer (SMP cluster) in which the number of processors in the compute nodes is greater than the total number of nodes, i.e. these are systems that have massive nodes or nodes in which there is a larger number of processors (at least 4) [Dongarra and van der Steen, 2012].

Federated clusters

Federated clusters are loosely coupled systems of computers or more specifically clusters. These systems have no interconnection network and can be architecturally quite different, but operate as a single resource. They are often referred to as clusters of clusters.

Grid

A grid is a type of computer system that enables the dynamic use of geographically distributed autonomous subsystems depending on their availability, capacity, performance and price. Grid computers are used to calculate large-scale computational problems such as N-body simulations, seismic simulations, atmospheric and oceanic simulations or protein folding (e.g. Worldwide LHC Computing Grid, Folding@home). At its core, a grid computer is a cluster in which the LAN is replaced with a WAN [Dongarra and van der Steen, 2012].

1.4.6 ccNUMA

Cache Coherent Non-Uniform Memory Access computers are most often included in the cluster family in the broadest sense of the term. The basic difference is in memory organization. On a hardware level, these computers are comprised of distributed SMP nodes. These nodes are connected using an interconnection network, similar to conventional clusters. The distinctiveness of ccNUMA computers is in the way SMP nodes access memory locations on other nodes. All memory, which is physically distributed, is addressed globally and all the data logically belongs to a single address space. Since the data is physically distributed and shared memory exists only at a logical level, access times can vary significantly, hence the inclusion of the NUMA in the name [Dongarra and van der Steen, 2012]. A schematic overview of a ccNUMA system is given in figure 1.19.

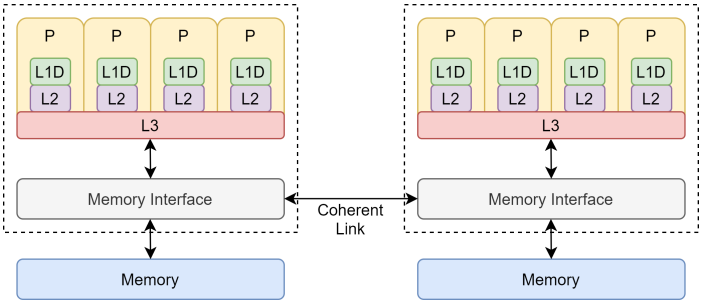


Figure 1.19 ccNUMA machine.

Term cache coherency infers that every variable must have a consistent value. The consistency of the variables is manifested in the fact that local changes in the variables, at a given node, are reflected globally [Dongarra and van der Steen, 2012]. Several elements ensure the consistency of the variables:

- **SBP (Snoopy Bus Protocol)** individual cache memories track the transport of variables to processors and refresh local copies of these variables.
- **Memory directory** is a special part of memory that enables tracking of all copies of variables as well as their validity.

An important performance metric when analyzing ccNUMA systems is the NUMA factor which shows the difference in latency when accessing data in local and remote memory (remote memory latency is typically considered for a furthermost node). Since all the data logically belongs to shared memory, ccNUMA systems can be considered SM-MIMD computers, i.e. MIMD computers with shared memory. However, since ccNUMA computers utilize software to create and maintain a shared memory space when compared to systems with shared memory at a hardware level (classic SM-MIMD), latency is higher by several orders of magnitude. Hence, although for a given program memory space might appear as shared, programmers must be aware of the computational costs writing and reading have when accessing distant memory locations.

1.5 Network architectures

University of Rijeka

A network that allows direct connection between each processor in a system, i.e. a fully connected network, is a superior connection design to any other option in terms of computational efficiency. However, implementation of such an interconnected system is rather complex, not to mention expensive, hence alternative topologies are commonly used. The performance of an interconnection network primarily depends on routing, flow-control algorithms and topology. Routing is the process of selecting an optimal path for traffic in a network. The process of managing the rate of data transmission between nodes is known as flow control, whereas network topology is the arrangement of various elements (such as communication nodes and channels) in an interconnection network. Among those mentioned, the network topology is the most troublesome, as the overall efficiency of the network depends on it, yet the employed design might not fit every user's needs.

Manufacturers of interconnected computer networks typically classify and advertise their products according to some of the key network performance parameters such as latency, bandwidth and scalability. A cost-effective system provides good throughput and low latency at an affordable price. Unfortunately, every network topology is not able to transmit memory requests quickly enough to be efficiently used for parallel computing. Interconnections have a major role in parallel computing hence a bottleneck in this aspect will significantly impair the ability to quickly perform computational tasks.

Interconnection networks can be classified into two main groups based on how the nodes in the network are connected [Trobec et al., 2020]:

- in **direct networks**, nodes are directly connected to all their neighbors.
- **indirect networks** utilize switches to connect nodes.

Indirect networks are typically more common due to their flexibility and can be further subdivided [Trobec et al., 2020]:

- **non-blocking** networks can always connect the source and destination regardless of the currently established connections.
- in **blocking networks**, established connections block the creation of new connections between the source and destination even though they might be idle.
- **blocking rearrangeable networks** are adaptable i.e. an established connection can be rearranged for a new connection to be established.

Latency

Latency represents the time that elapses from the moment the message is sent to the moment when the message reaches its destination [Trobec et al., 2020]. The total latency time can be divided into several types:

- latency at the MPI software level represents the time from the moment the send command was issued to the moment of the execution of the received command and it is measured by the ping-pong standard test.
- application-level latency is the time from a call to the communication library

function to the moment when the receive function on the receiver side excites the MPI.

- latency at the hardware level.
- latency at the interconnection level.

Latency is measured in milliseconds and is often interchangeably referred to as a ping rate.

Bandwidth

Bandwidth is the theoretical maximum data transfer rate through one channel in a given time [Trobec et al., 2020]. In layman's terms, it measures the maximum capacity. Commonly employed units for bandwidth are **Mbytes/s (MB/s)** and **Gbytes/s (GB/s)** while for serial channels it is given in **Mbit/s (Mb/s)** or **Gbit/s (Gb/s)**.

Throughput

The amount of data that is transmitted through a communication link within a period is called throughput. It is commonly referred to as the effective data rate or payload rate. Bandwidth and throughput differ due to various technical issues, latency, packet loss, etc.

Interconnection network parameters

Interconnection networks are typically described using graph theory, where a network is modeled as a graph $f(k,n)$, which consists of k communication nodes and n communication links (channels) between said nodes. This approach enables the definition of parameters that can be used to classify networks [Trobec et al., 2020]. These parameters include:

- **node degree** d is the number of neighbors of a node in a network.
- **regularity** is the property that states that all nodes have the same node degree.
- **path** represents a path from the source to the destination node.
- **hop count** is the number of nodes data traverses along its path.
- **diameter** l_{max} is the maximum distance between two nodes in the network i.e. maximum hop count.
- **complexity** is the total number of connections or switches in the network.
- **connectivity** represents a minimum number of broken connections that would cause a system crash.
- **scalability** ability of a network to withstand degradation due to the increase in network size.

1.5.1 Bus

The bus is the simplest network topology. All processors are connected to a single pathway, a bus, which can transfer a single piece of information at any given time. Consequently, processors must usually wait in queue for the bus to become idle before sending or receiving new information. A schematic overview of a bus network topology is given in figure 1.20.

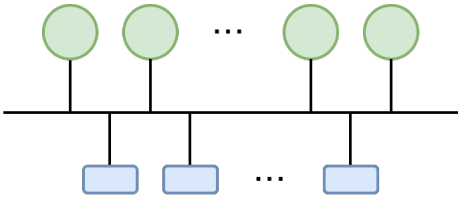


Figure 1.20 Bus network topology.

Implementation of the bus topology is rather simple and inexpensive. When it is used to connect a smaller number of processors, the network can be effective with the efficiency further improved by including local caches [Trobec et al., 2020]. For larger systems, the contention is significant and makes the network ineffective.

1.5.2 Ring

The ring is one of the oldest network topologies. Processors are arranged in a consecutive, linear fashion, in a ring, with every processor having only two neighbors [Trobec et al., 2020]. Ring network topology is shown in figure 1.21.

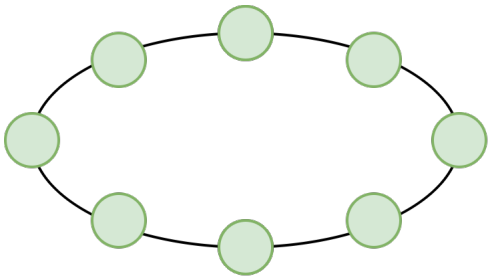


Figure 1.21 Ring network topology.

1.5.3 Star

Star network topology is a centralized design where all the communication between the processors is routed through a central hub. The hub is tasked with routing the traffic and based on the number of processors in the network this can hinder the performance [Gebali, 2011]. The network schematic is given in figure 1.22.

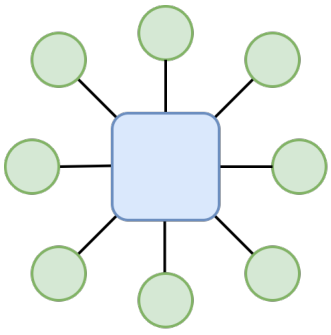


Figure 1.22 Star network topology.

1.5.4 Mesh

These types of networks are currently commonly employed in systems with large numbers of processors and are considered successors to the older hypercube topology.

2D mesh and 2D torus

Processors in the 2D mesh are arranged in a rectangular manner so that every member can be defined through its (i,j) label. Every processor has four neighbors, apart from those on the edges of the network [Trobec et al., 2020]. If we assume that there are n nodes in the network, the parameters of the network are as follows: $d \approx 2\sqrt{n}$, complexity $2n$ and connectivity 2.

2D torus design nullifies the topological deficiencies of the 2D mesh by arranging processors in a toroidal structure. Consequently, processors on the edges have the same number of neighbors as a typical centrally located processor in a 2D mesh. This network topology is symmetric [Trobec et al., 2020].

3D mesh and 3D torus

3D meshes are a logical extension of their 2D counterparts. Functionally they are the same. The primary difference is in the number of neighboring nodes which in three dimensions is six. Similar deficiencies on the edges are apparent and are mitigated in 3D torus design [Trobec et al., 2020]. For n nodes, parameters of the network are: $d = 1...\sqrt{n}$, complexity $2n$ and connectivity 4. An overview of different 2D and 3D topologies is given in figure 1.23.

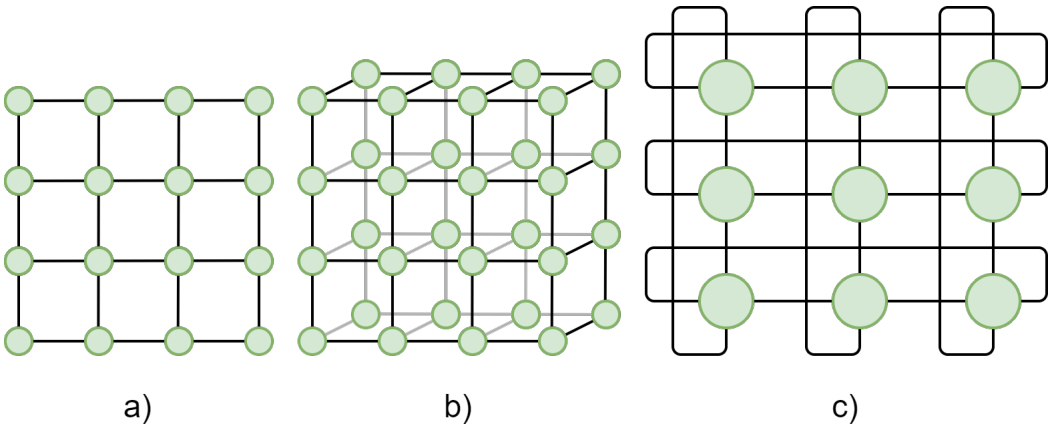


Figure 1.23 2D and 3D meshes: a) 2D mesh, b) 3D mesh, c) 2D torus.

1.5.5 Hypercube

In an effort to better balance interconnection network quality and price, networks have been developed based on the principle of the so-called hypercube. Each node in a hypercube is connected to $b = log_2 n$ neighbors, therefore the network can connect $n = 2^b$ nodes [Trobec et al., 2020]. The dimensionality of the cube is n, hence they are commonly called n-cube networks.

One of the key benefits is their symmetric nature, which means that the network appears the same from every node, hence no special treatment for nodes is needed. Moreover, it is highly reliable as it provides n alternative paths (disjoint paths) between any two nodes, thus in the case of a failure of a given path, the network would

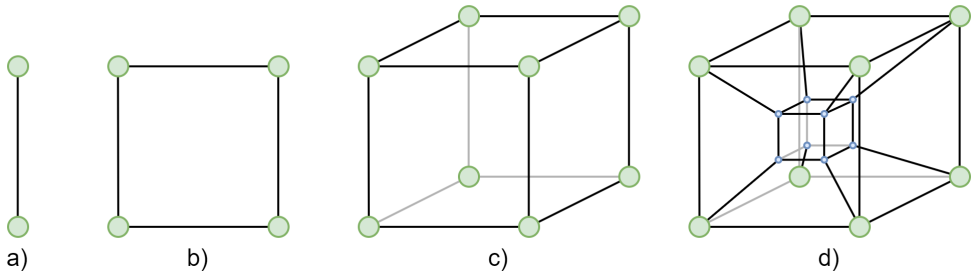


Figure 1.24 Principle of the hypercube network topology: a) $d = 1, l_{max} = 1$, b) $d = 1...2, l_{max} = 2$, c) $d = 1...3, l_{max} = 3$, d) $d = 1...4, l_{max} = 4$.

continue to function normally. Two-dimensional meshes and trees can be embedded in a hypercube in such a manner that the connectivity between neighboring nodes remains consistent with their definition. For parallel systems with large numbers of processors, $2D$ and $3D$ network topologies (e.g. torus) are the current state-of-the-art topologies, although in recent years hypercubes have seen a resurgence.

1.5.6 Fat tree

When connecting a large number of nodes, the so-called fat tree topology is extremely popular. This topology is based on the known structure of a tree. Congestion typically occurs near the root of the tree due to the concentration of messages that traverse through higher levels before descending to the target nodes. The fat tree solves this problem by introducing additional connections at those tree levels, thus increasing the bandwidth [Trobec et al., 2020].

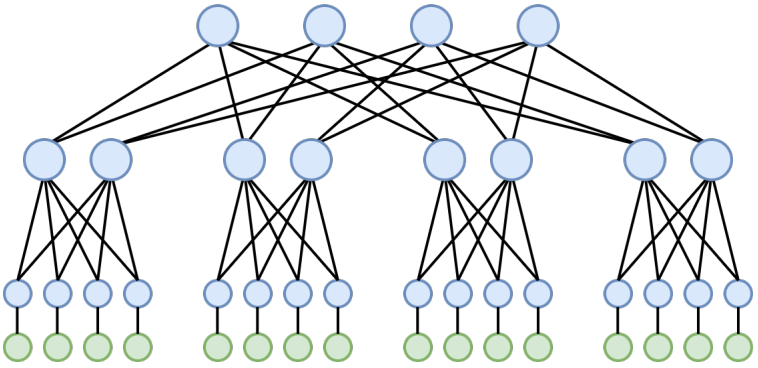


Figure 1.25 Fat tree topology.

The term n-level fat tree defines a fat tree structure in which the number of connections at the level closer to the root is n times higher than at the level above.

A simpler implementation is a binary tree network where each switch in the network has three links and the processors are located at the top [Trobec et al., 2020].

1.5.7 Fully connected network topology

Each node in a fully connected network (direct network) is directly connected to all other nodes. If we assume that there are n nodes in the network, there are in total $n(n-1)/2$ connections. The diameter of this network is 1. The network does not scale well (requires too many connections) and is thus used only sometimes, for smaller

clusters or specific purposes (e.g. military applications).

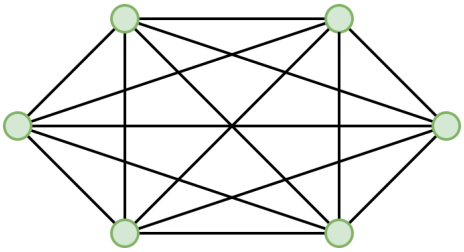


Figure 1.26 Fully connected network topology.

1.5.8 Crossbar network

A crossbar network is built upon a simple two-dimensional grid of switches. Design connects n inputs and n outputs (i.e. processors) and requires n^2 switches. Analogously to fully connected networks, any two members of the network can directly communicate, if the required ports are free. In total, n concurrent connections are possible. Communication is achieved through a change in the switch's state. Consequently, these networks are typically referred to as dynamic networks. If the port(s) are occupied, all subsequent communication has to wait for the port(s) to be free. Due to this, crossbar networks include arbiters that regulate queues [Gebali, 2011]. For a MIMD system with a large number of processors, this network topology would present a technologically too complex design with significant underlying costs. On systems with a relatively small number of nodes, however, especially if those nodes possess significant computing power, e.g. parallel vector machines, a fully connected crossbar network is an obvious choice. Crossbar networks are commonly used in high-performance small-scale shared-memory multiprocessors, routers for direct networks and as a fundamental component of large-scale indirect networks. A schematic of a crossbar network is given in figure 1.27.

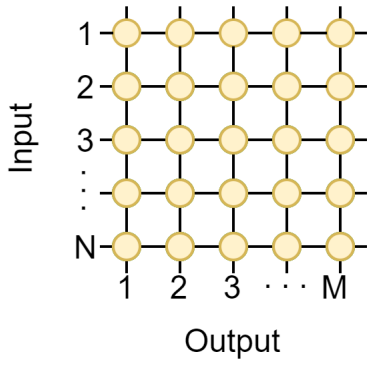


Figure 1.27 A crossbar network.

1.5.9 Multistage interconnection network

One of the solutions to the crossbar topology problem is multistage interconnection networks. They belong to the dynamic subgroup of networks. Inputs and outputs are connected through a stage (set) of switches. These switches are fewer in number, hence a single-stage design cannot connect all the outputs and inputs. By cascading the single-stage switches, all the inputs and outputs can be connected, with significant

savings [Gebali, 2011, Trobec et al., 2020]. Level 1 switches in this design, unlike single-stage topology, are connected to level 2 switches, etc., instead of being directly connected to the outputs.

A typical example of a multistage interconnection network is the Omega network (Ω). When connecting $n \cdot n$ sized Omega network, there are in total $log_2 n$ stages with $n/2$ switches per stage. In total $n/2 \cdot log_2 n$ switches are needed which is substantially fewer than for a crossbar network (n^2).

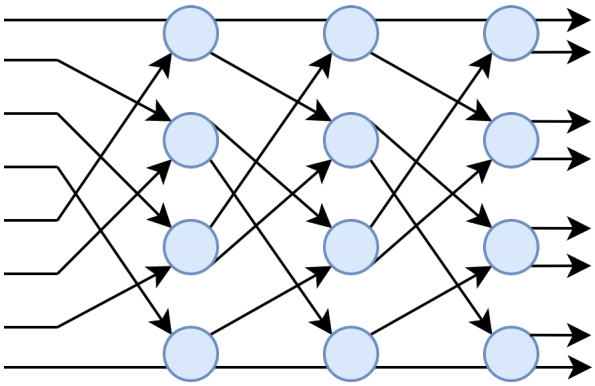


Figure 1.28 Omega network.

1.5.10 Concluding remarks

The most common interconnection networks used in the design of highly parallelized machines are based on the following concepts: hypercube, fat-tree, torus, and full crossbar. It is critical to emphasize the growing importance of interconnection networks, which is a direct result of the rapid increase in the number of processors per parallel computer. Consequently, interconnection networks have been designed to accommodate median (average) congestion rather than providing adequate bandwidth to accommodate the so-called worst-case loads.

The current design philosophy implies traffic congestion, which is deemed acceptable due to cost savings. Consequently, new branches of computer science are in development, that deal with the so-called congestion management. Congestion management presupposes modern, adaptive networks, with reroute strategies, and the field as a whole is extremely appealing from an engineering and mathematical standpoint.

1.6 Current trends

University of Rijeka

Even though semiconductor sizes are continuously being shrunk, with each generation physical production limits are being tested. Heat and heat-associated frequency limits are a persistent problem. Physical limitations have directed the development of processor architectures to immensely parallel designs which has in turn led to significant advancements in the performance of massively parallel architectures. Multicore systems (dozens or even hundreds of cores) with simultaneous multithreading and asymmetric execution are of increasing interest. Consumer GPUs as of 2020 have upward of 10000 cores and are indispensable as accelerators in complex HPC systems. Existing massive core designs are slowly being replaced by simple power-efficient cores (ARM), which, although comparatively worse-performing, offer better value both in terms of initial investment as well as general performance/$. Overall performance has steadily improved and HPC systems have crossed the exascale barrier in 2022.

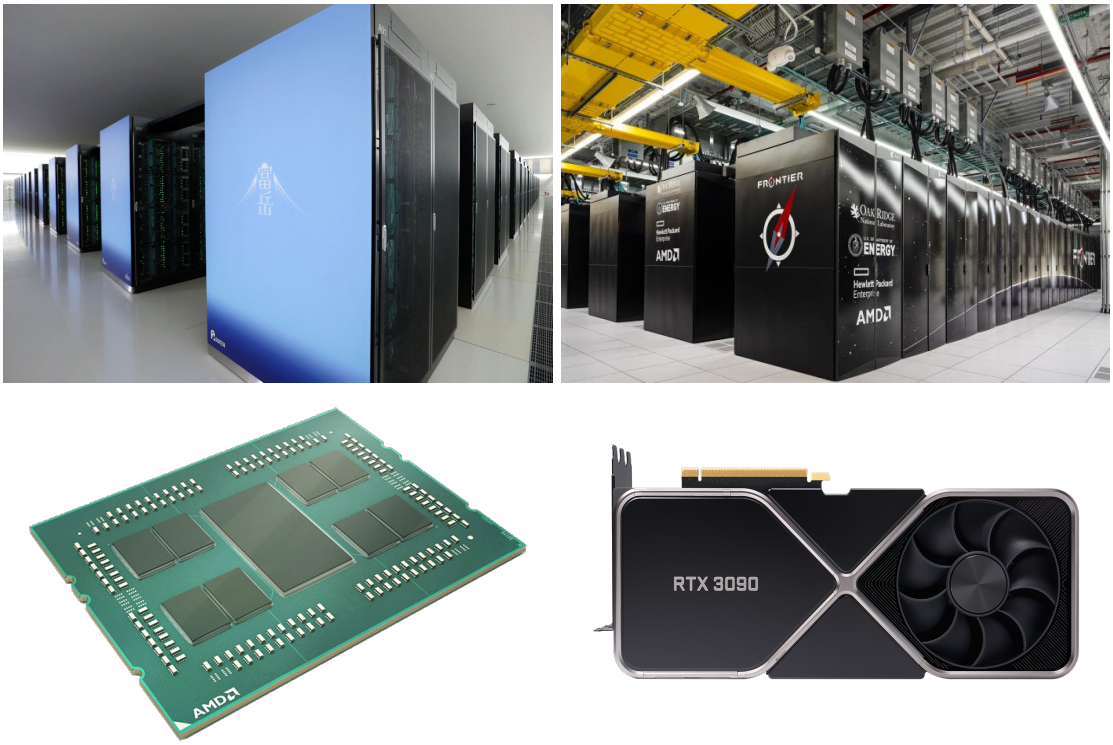
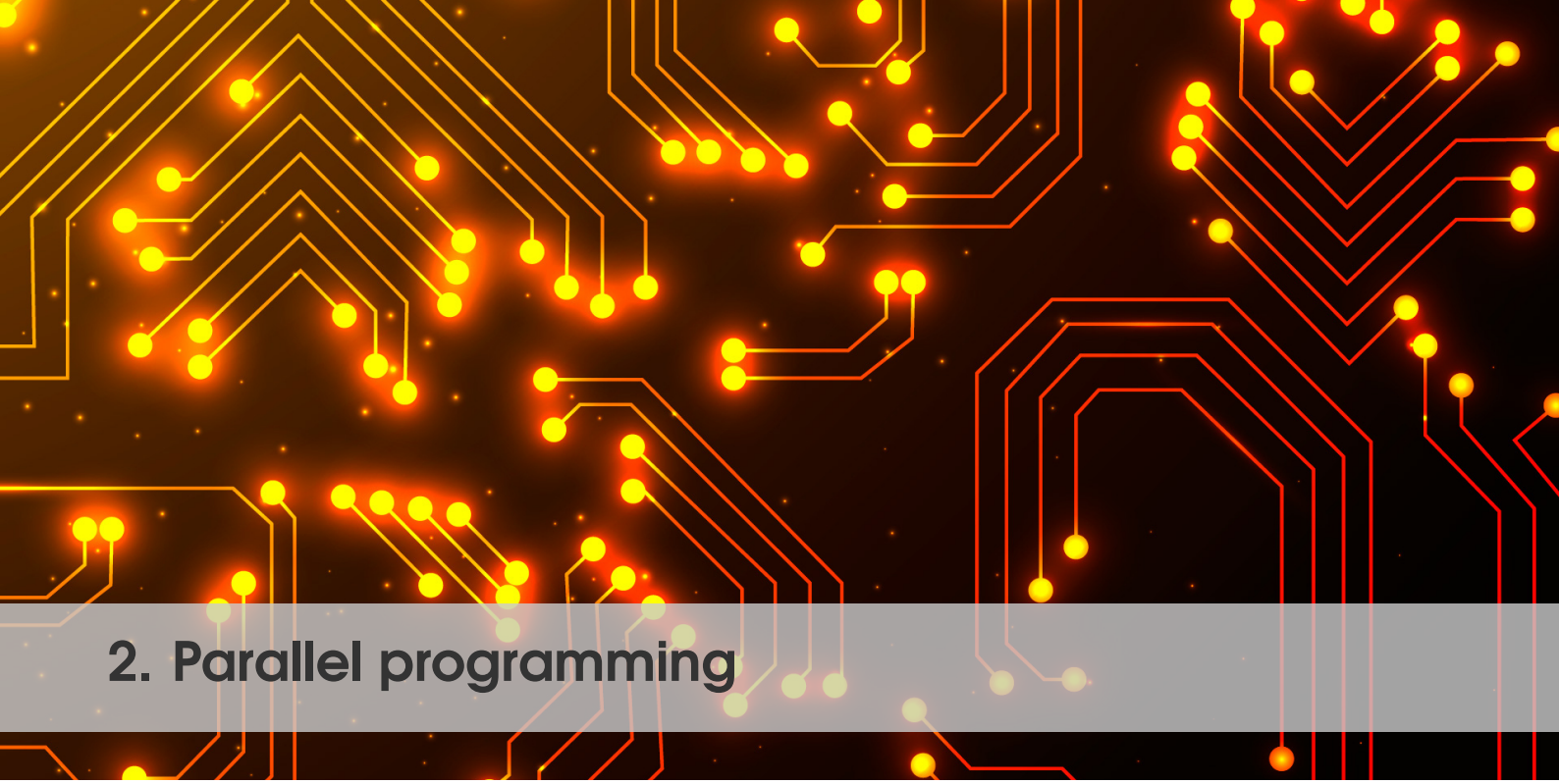


Figure 1.29 a) Fugaku - ARM-based supercomputer with over 7 million cores. b) Frontier - first supercomputer to cross 1 EFlop/s. c) AMD Epyc CPU - processor built using chiplet design with ccNUMA employed to schedule tasks. d) Nvidia RTX 3090 - consumer grade GPU with 10496 CUDA cores.



2. Parallel programming

Parallel programming concept
Parallel program analysis
Parallel programming models
OpenMP and MPI
GPU computing

2.1 Parallel programming concept

University of Rijeka

Knowledge of parallel computer architectures, memory organization, topology and interconnection network characteristics is necessary in order to be able to adequately develop a code/program and/or install a specific application. By following the parallel computer classification principle defined by Flynn, a proper programming approach can be adopted, in order to optimally utilise the parallel architecture.

Parallel programming concepts and models will be assessed on exampled i.e. simulation of an engineering problem (dam break). Numerical calculations are assumed to be conducted on a four-core parallel computer. Different parallel strategies will be addressed depending on the available computer architecture and software. The efficiency of the parallelisation will be analysed and the appropriate program parallelisation model recommended.

2.1.1 The dam break problem

As previously asserted, parallel programming will be assessed on a dam break test case. The computational domain is comprised of two water tanks separated by a dam. The water level is higher in the left tank. Figure 2.1 show the entire discretized domain of the instantaneous dam break test case. Numerical simulation can be performed using the finite volume method, finite element method, finite difference method, etc. Due to the nature of the problem, it is possible to define the data as a matrix $\mathbf{A}(1:n,1:m)$.

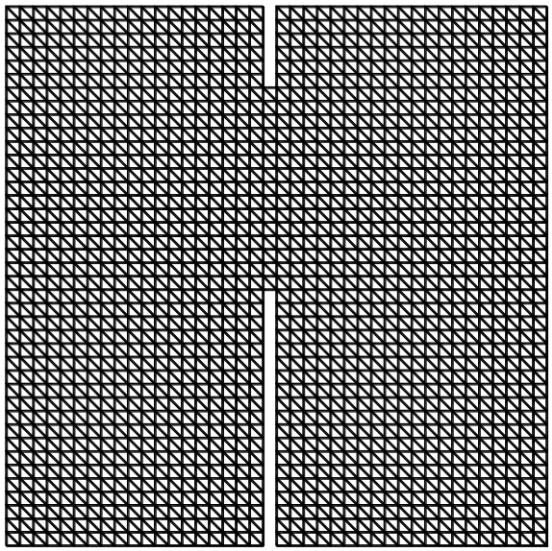


Figure 2.1 Numerical domain of a dam break test case.

Let's assume that for a given problem it is necessary to solve a system of partial differential equations. Furthermore, let's assume that at some point it is also necessary to calculate the values of matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and vector $\mathbf{d}$, with the $\mathbf{x}$ being the unknown vector, although depending on the employed numerical scheme, this could vary. In matrix form this can be written as:

$$(\mathbf{A}+\mathbf{B}+\mathbf{C})\cdot\mathbf{x}=\mathbf{d} \tag{2.1}$$

The solution to this problem can be attained with a sequential code such as that given in 2.1.

Algorithm 2.1 Dam break sequential code algorithm.

```
declare d(n),A(n,m),B(n,m),C(n,m)
do i=1,n
d(i) = ... → get d
do j=1,m
A(i,j) = ... → get A, B, C
B(i,j) = ...
C(i,j) = ...
end do
end do
x(i) = ... → solve system of equations
```

When formulating a parallelization strategy for a given problem i.e. dam break, it is important to consider both processing and memory requirements. Parallel programming, therefore, relies on:

- appropriate job distribution to processors.
- appropriate data distribution to processors (provided that a DM machine is used).

2.1.2 Job and data distribution

Fundamental parallelization strategies that can be utilised depending on the physical/numerical problem and parallel architecture type are job distribution, data distribution, domain decomposition and in some cases functional decomposition.

Job distribution

The strategy behind the job distribution when calculating expression 2.1 is shown in code snippet 2.2. Job distribution approach is shown for matrix A and 100 iterations. Calculations for other matrices and vectors can be implemented in a similar manner.

Algorithm 2.2 Dam break job distribution.

```
compute A, iterations 1–25   → assign to processor 1
compute A, iterations 26–50  → assign to processor 2
compute A, iterations 51–75  → assign to processor 3
compute A, iterations 76–100 → assign to processor 4
```

According to this principle, the total calculation volume is distributed among available processors. This mode of parallelization implies that all of the n iterative steps to be carried out on p processors are distributed so that each processor calculates only n/p iterations i.e. when utilising four processors, the first quarter of all iterations is performed on the first processor, the second quarter on the second, etc. Such a division of iterative jobs per processor is possible only if calculations can be performed independently i.e. regardless of the results from other processors.

Data distribution

Data distribution strategy that would solve the problem outlined in equation 2.1 is given in snippet 2.3. Given example of the data distribution among four processors for matrix **A** should be implemented for the remaining matrices and vectors as well.

Algorithm 2.3 Dam break data decomposition.

```
A(1:20,1:50)   → data assigned to processor 1
```

$\mathbf{A}(1:20,51:100) \rightarrow$ data assigned to processor 2
$\mathbf{A}(1:20,101:150) \rightarrow$ data assigned to processor 3
$\mathbf{A}(1:20,151:200) \rightarrow$ data assigned to processor 4

With this type of parallelization, identical instructions (operations) are performed on all processors although on different sets of data. This approach is typical for a MIMD architecture, but can also be used on a SIMD machine.

Domain decomposition

In engineering practice, the most common parallelization strategy is the domain decomposition strategy shown in figure 2.2. Domain decomposition is accomplished at a higher level i.e. one level closer to the physical model, hence the entire approach is less abstract.

For a four-core parallel machine, the entire computational domain is decomposed i.e. split into four parts, each of which is assigned to its processor. Apart from being assigned a subdomain, each processor must be given, prior to or at runtime, a specific set of instructions as well as provided with boundary cell data innate to the neighboring domain parts. The amount of data that must be exchanged between the processors which are assigned neighbouring parts of the domain depends on the method of the domain decomposition i.e. whether the domain segments overlap (e.g. Schwarz method) or do not overlap (e.g. Schur complement method).

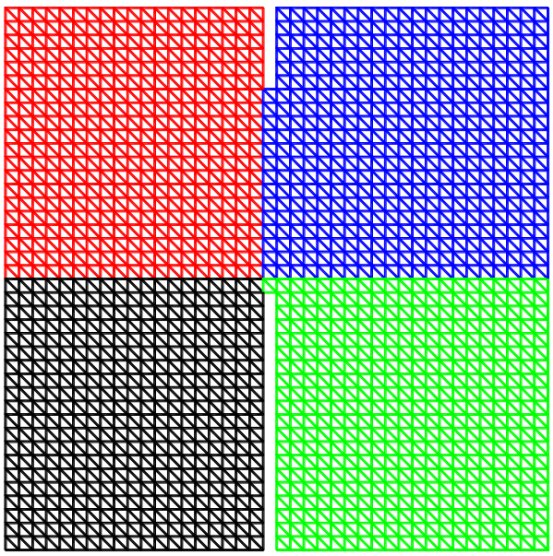


Figure 2.2 Dam break domain decomposition using four processors.

Figure 2.3 depicts a decomposed domain. Let's assume that the cell-centered finite volume method is used. Highlighted zone shows elements in the neighbouring subdomains. In order to calculate a new value for u_i, values at neighbouring cells, including u_{i+1} which is in the adjacent subdomain, are needed. Depending on the stencil of the numerical scheme, values in multiple adjacent rows might be needed as well. Communication between processors is a major limiting factor in the overall performance of a program in this case, hence proper decomposition is of utmost importance.

Domain decomposition is typically achieved with an automatic algorithm, but can also be performed directly i.e. manually, for simpler domains. Algorithms for numerical domain decomposition commonly employ graph theory. In graph theory,

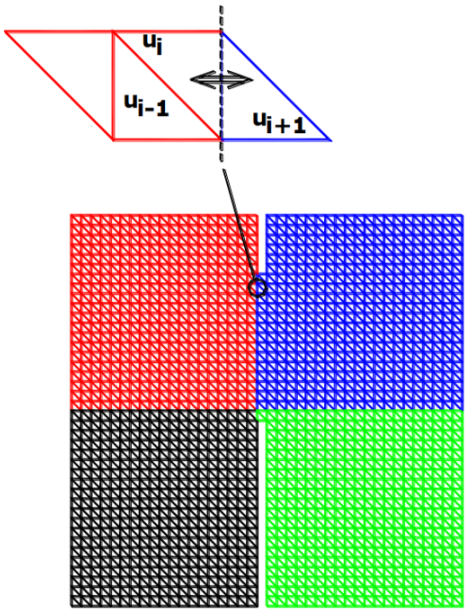


Figure 2.3 Communication between boundary elements of subdomains.

the domain is depicted with a set of vertices and edges $\mathbf{G}(vertices, edges)$, which are to be divided into k equal or approximately equal segments in a way that minimizes the number of intersections between the edges and the dividing line. This concept is depicted in figure 2.4.

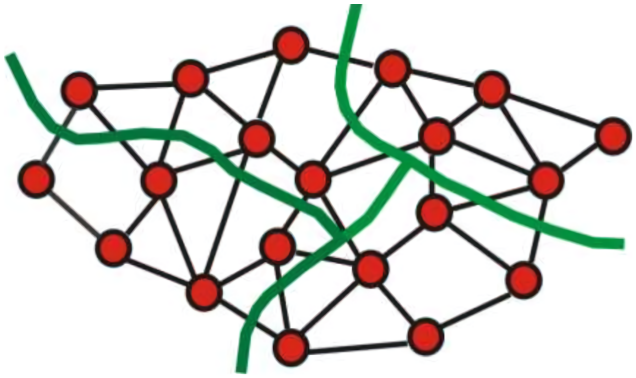


Figure 2.4 Decomposition using graph theory.

From a computational standpoint, every vertex (point) in figure 2.4 represents a computational load while the edges correspond to communication. The goal is to minimize the communication (the least amount of intersections with the dividing line) and appropriately distribute the load amongst the processors. This means that for certain problems appropriate decomposition is not an "equal" split approach. Balancing is particularly important if utilised processors have different performance or the communication between them is inconsistent.

The majority of domain decomposition algorithms can be divided into the following groups:

- simple algorithms.
- inertial method.

- spectral distribution method.
- Kernighan-Lin and related methods.
- multilevel methods.

Simple algorithms

The most common simple algorithm is the linear method. Vertices are assigned to individual processors according to their indices in the original graph. This simple scheme often gives surprisingly good results as the grouping of the vertices has typically already been done during the initial indexing. In addition to the linear method, the random method and scattered scheme can be utilised. The random method assigns vertices to processors randomly whereas the scattered scheme assigns vertices in the same manner in which the cards are dealt in a card game.

Inertial method

The inertial method is relatively simple and fast. In addition to the graph data, it also requires geometric coordinates of every vertex. Vertices are seen as "heavy" points, and their non-negative weight is determined in proportion to the computational effort required to calculate the new state of a given point. Said computational effort is approximately proportional to the number of points within the distance h of a given point.

Spectral distribution

The spectral distribution uses the eigenvalues of the matrix generated from the graph to define the distribution.

Kernighan-Lin method

The Kernighan-Lin method (KL) is in essence a local optimization strategy. Vertices are swapped between different sets in order to minimize the number of graph edges intersected by the dividing line. This method is typically not suitable for large graphs, however, The Multilevel KL method can be used as an alternative.

Multilevel methods

All multilevel methods typically include three common phases:

- coarsening phase.
- partitioning phase.
- refining phase (uncoarsening).

In coarsening phase, smaller and coarser graphs are extrapolated from the original graph, with each coarser graph created through contraction of the edges of the original graph, according to a set methodology. As the edges are contracted, their two limiting vertices are combined into a new vertex. These new vertices and edges contain the "weight" of the original vertices and edges from which they were built, hence, in a way, information about the initial graph is preserved. The partitioning phase of the coarsest graph is achieved by one of the previously mentioned methods e.g. the spectral method. During the refining phase, the graph is backward-reconstructed to its original state with the partitioning repeated for every level of refinement. The KL method is commonly used to re-partition each intradivision. The main phases of a multilevel method are shown in figure 2.5.

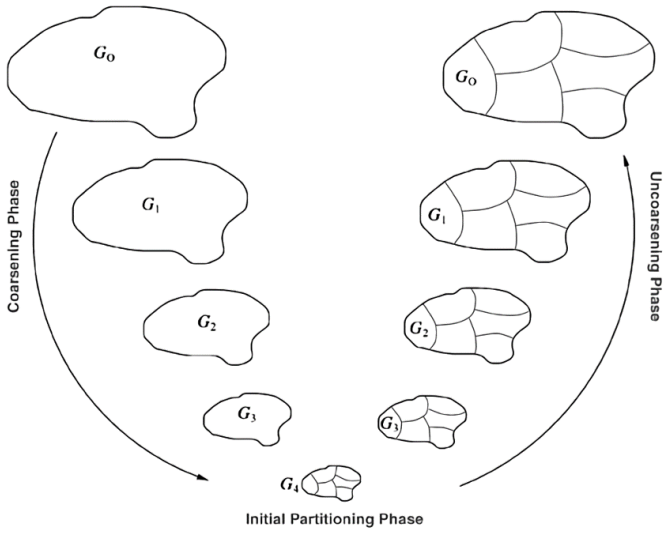


Figure 2.5 Phases of a multilevel method.

The most popular algorithms for domain decomposition that utilise described principle are METIS, its parallel version PARMETIS and the CHACO algorithm. In addition to the decomposition, these algorithms can also improve the quality of the existing decomposition, provide dynamic re-decomposition of adaptive grids and manage load balance after each modification of the adaptive network.

Functional decomposition

Functional decomposition is the least common decomposition approach, mainly due to specific implementation requirements. The fundamental prerequisite is the ability to perform different operations on each processor or compute node. Hence, only specific types of computers can be utilised. This method of parallelization can be employed to calculate complex physical models that consist of several relatively independent lower level physical models. For example, when calculating a complex climate model which consists of the atmospheric model, ocean model, Earth's surface model and the hydrological model, it is possible to separate said segments at a lower-level, provided they use mostly independent data. Consequently, a single or a group of processors could be tasked to calculate each segment of the climate model separately. If there is a significant overlap i.e. data dependence between sub-models, excessive communication and data exchange hinder the performance and other models should be used.

Functional decomposition can be employed for numerical analyses of the flow around aircraft, with potential flow models used in one part of the domain and Navier-Stokes equations in other. Similarly, hybrid domains which combine Lagrangian particle methods and the Euler approach can be efficiently decomposed.

2.2 Parallel program analysis

University of Rijeka

The quality of the parallelized computer code, regardless of the parallelization model, can be quantified with three distinct parameters:

- speedup.
- efficiency.
- scalability.

Speedup is the most important measure of the quality of parallelization. Efficiency and scalability coefficients are typically used as supplementary indicative parameters.

Let's define $T(p,N)$ as the time required to solve a problem of a size N on p processors, and $T(1,N)$ as the time required to solve the same problem on a single processor. Speedup $S(p,N)$ can consequently be calculated as:

$$S(p,N) = \frac{T(1,N)}{T(p,N)} \tag{2.2}$$

Given expression defines the degree of speedup i.e. how much faster is the code when executed on N processors compared to a single processor? In an ideal case:

$$S(p,N) = p \tag{2.3}$$

Real codes, however, are not ideally parallelized, hence it is necessary to determine the efficiency i.e. how close are we to the ideal speedup:

$$E(p,N) = \frac{S(p,N)}{p} \tag{2.4}$$

Occasionally, when measuring the quality of the parallelized code, speedup might be larger than the theoretical ideal value, p, with efficiency $E(p,N) > 1$. This anomaly is related to the problem size. When measuring the time needed for a single processor to solve a given task, said task is typically scaled down in order to fit in the memory which is addressable by a processor. Consequently, when the same task is run on multiple processors, e.g. 512, individual batches allocated to the processors might be small enough to fit in the processor's cache, which makes the computational part extremely fast and leads to unrealistic speedups. In order to eliminate these false results or at least provide insight into potentially suspicious speedups, the scalability $Sc(p,N)$ coefficient is defined:

$$Sc(p,N) = \frac{N}{n} \tag{2.5}$$

where N represents the size of the original, and n size of the scaled problem. The scalability coefficient is relevant and calculated only if:

$$T(1,N) = T(p,N) \tag{2.6}$$

For a computer with p processors, the scalability coefficient defines the size of a problem that can be calculated in the same amount of time that is required for a single processor to calculate a similar but smaller task.

At the beginning of this section we stated that both memory and processors are essential resources that should be considered when parallelizing a given problem. The following notes briefly summarize key aspects of load and data distribution goals as they relate to these resources:

- Computational load should be distributed across processors in a manner that minimizes waiting and synchronization times i.e. processors should be assigned jobs that are equally computationally demanding so that they have approximately similar calculation times. This is typically achieved through load balancing.
- Data should be distributed with regards to the overarching computer architecture so that the communication overhead/access times are minimal.

2.2.1 Amdahl's law

Theoretically, the speedup from parallelization should be a linear function of the number of processors. This, however, is not the case since most algorithms cannot be fully parallelized. Speedup is typically near-linear for small numbers of processors and then flattens out and assumes a constant value for large numbers of processors.

Amdahl's law predicts the theoretical maximum speedup of a code as a result of the increase in the processor count. The speedup is lower than in an ideal case since it is limited by the sequential segment of the program. If we define the execution time of a sequential part of the code i.e. the part of the code that cannot be parallelized as s, then the parallelized segment equals q. The total execution time for a problem of a size N on a single processor is the sum of sequential and parallel parts of the code:

$$T(1,N) = s + q \tag{2.7}$$

For p processors the total time equals:

$$T(p,N) = s + \frac{q}{p} \tag{2.8}$$

Speedup according to Amdahl's law can therefore be calculated as:

$$S(p,N) = \frac{T(1,N)}{T(p,N)} = \frac{s+q}{s+\frac{q}{p}} \tag{2.9}$$

Let's assume, for simplicity, that the total time equals unit time. Consequently $s+q=1$. Amdahl's law can hence be formulated as:

$$S(p,N) = \frac{1}{1-q+\frac{q}{p}} \tag{2.10}$$

It is evident that with the increase in processor count i.e. when $p \to \infty$ speedup is limited by the sequential part of the code:

$$S(p,N) < \frac{1}{1-q} \tag{2.11}$$

The asymptotic behavior of the speedup curves is therefore, according to Amdahl's law, understandable and expected. Speedup curves as functions of the number of processors for different cases are shown in figure 2.6. If we consider the case that has been run on 1000 processors with the sequential part of the code equal to 0.1%, the

overall speedup is $S(1000, N) = 500$ and efficiency $E(1000, N) = 0.5$. This means that for heavily parallelized codes sequential parts should be reduced to a minimum, in order to achieve appropriate efficiency and speedups, which is typically the case for large test cases with extensive databases.

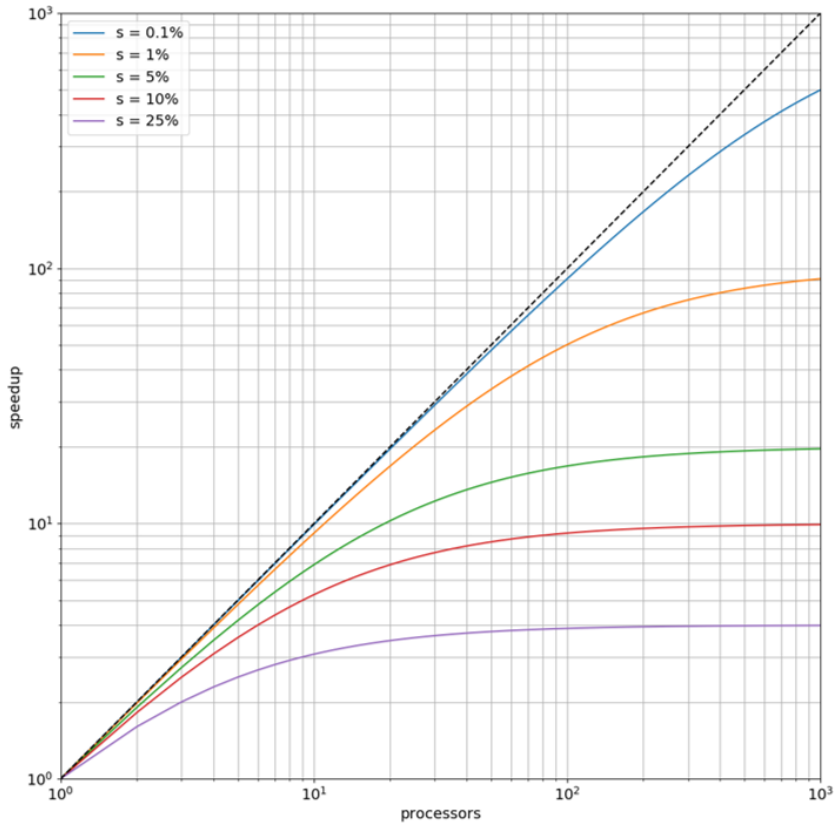


Figure 2.6 Amdahl's Law.

2.2.2 Gustafson's law

One of the key assumptions in Amdahl's law is that the fraction of the parallelizable code is constant regardless of the workload size i.e. workload is assumed to be fixed. This view, however, is pessimistic and not true for large problems. Gustafson's law assumes that the execution time is constant and gives the theoretical speedup that can be achieved with the increase in the number of processors.

Let's define the execution time of a sequential part of the code as s and the parallelized segment as q. For parallelized code utilising p processors, we can write:

$$T(p, N) = s + q \tag{2.12}$$

For a single processor, the total time increases according to:

$$T(1, N) = s + q \cdot p \tag{2.13}$$

Let's now calculate the speedup according to these assumptions:

$$S(p, N) = \frac{T(1, N)}{T(p, N)} = \frac{s + q \cdot p}{s + q} \tag{2.14}$$

If we apply the simplification $s + q = 1$, theoretical speedup now becomes:

$$S(p, N) = 1 - q + q \cdot p \tag{2.15}$$

Speedup curves as functions of the number of processors for different cases according to Gustafson's law are shown in figure 2.7.

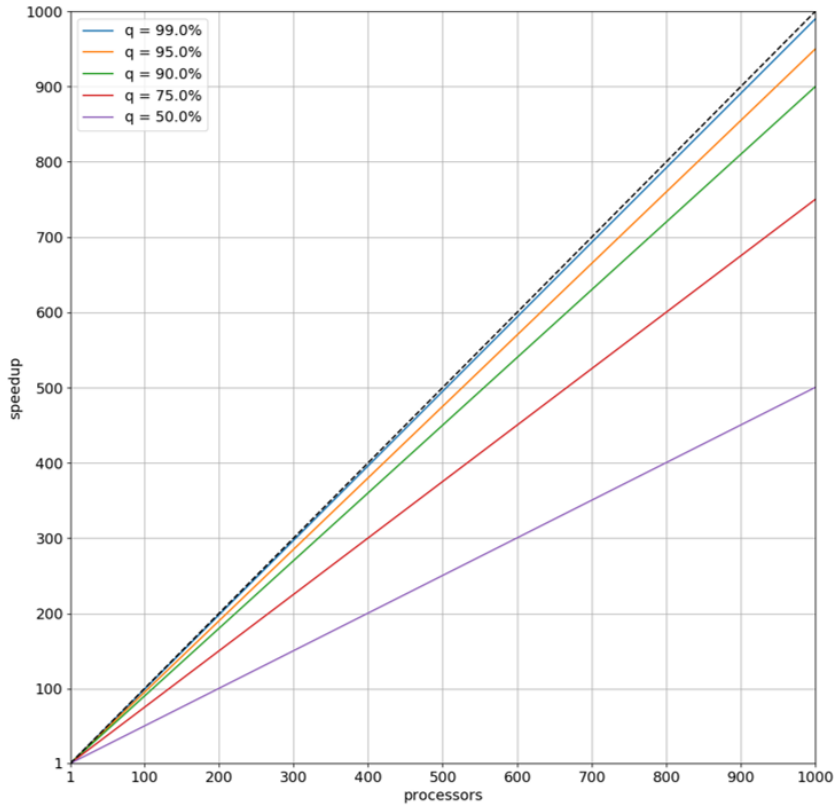


Figure 2.7 Gustafson's Law.

2.2.3 Reduced efficiency and its causes

The main causes of reduced efficiency of a parallel program that cannot be completely avoided are communication and synchronization overheads. Communication presupposes the synchronicity of the processes that communicate. In case of inappropriate load distribution, the processor that executes the instructions faster and reaches the synchronization point sooner will have to wait until a slower process reaches the same point in order to, e.g. exchange or update data. Sync requests, however, do not have to necessarily be just for communication.

For problems and applications that rely on communication between the processors, data replication might be a solution. Replication implies that the data assigned to a given processor is already distributed to another processor. Concept induces additional computational load on the system as a whole due to the duplication (multiplication) of identical operations which are already executed on different processors. This means that, in order to minimize the communication overhead, additional resources i.e. memory and computational tasks can be utilised. Depending on the available resources, this might not be an option, or not feasible regardless of the resources, hence it is necessary to assess which approach is more suitable.

Communication overhead and data replication can be best understood through an example. HITACHI SR8000 is a parallel system with a theoretical performance of 1 GFlop/s per processor while the interconnection latency is 6 μs. Interconnection bandwidth is 300 MB/s. The latency when transferring a single bit of data is identical to the time required to perform 6000 instructions. Let's assume that we want to transfer a domain segment containing $20 \cdot 20$ elements, with each element described with 5 distinct physical values (e.g. velocity, pressure, etc.). This means that we need to transfer 2000 variables, each in double precision format. In total, 16KB of data needs to be transferred. Given the interconnection bandwidth, we can calculate the time needed to transfer said data, which is approximately equal to 53300 operations on a processor. Even if we neglect latency, it is evident that the communication overhead has a significant effect on the overall processing time.

2.3 Parallel programming models

University of Rijeka

Parallel programming models are closely related to the computer architecture and implemented parallelization strategy i.e. job distribution, domain decomposition, etc. Typically three distinct models are discussed:

- Parallel Programming on a SM System (Open Multi Processing - OMP).
- Parallel Programming on a DM System (Message Passing Interface - MPI).
- Parallel Programming Using the Data Distribution Strategy.

2.3.1 Parallel programming on a SM system

This type of parallelization is used on systems that utilise either physical or logical (ccNUMA) shared memory concept. Parallelization can be typically achieved by inserting specific commands into the sequential code which delineate parallel code segments. Loops are usually parts of the code that are parallelized. Communication and data distribution are typically either not addressed or, depending on the compiler, can not be managed by the programmer. As of 1997. a standardized set of compiler commands, libraries and system variables for parallel programming of SM computers exists called OpenMP or OMP (Open Multi Processing). OMP parallel commands can be easily integrated into Fortran or C/C++ code.

One important feature of the OMP protocol is its adaptability to different programming concepts i.e. same code can be used in both sequential and parallel mode. When compiling a sequential code, OMP commands are interpreted as comments. In parallel mode, commands can be recognized by corresponding compilers: !$OMP in Fortran, c$OMP in Fortran77 and #pragma omp in C/C++. Implementation of OMP protocol is shown on a previously defined sequential code (2.1):

Algorithm 2.4 Sequence of code parallelized using OMP protocol.

```
1   declare d(n),A(n,m),B(n,m),C(n,m)
2   !$OMP PARALLEL DO
3   do i=1,n
4   d(i)=... → get d
5   do j=1,m
6   A(i,j)=... → get A, B, C
7   B(i,j)=...
8   C(i,j)=...
9   end do
10  x(i)=... → solve system of equations
11  end do
12  !$OMP END PARALLEL DO
```

OMP protocol, as seen in code snippet 2.4, uses the so-called fork-join model i.e. code at a given point branches and begins parallel execution and at a subsequent point joins and resumes sequential execution (Figure 2.8).

Described method of parallelization, which is in essence fairly automatic, provides great freedom, flexibility and ease of parallelization. Ease and freedom, however, are only apparent as programmers must be familiar with the sequence of the code that is to be parallelized since there is a chance that parts of the code that have dependencies or parts that can not be parallelized for other reasons are marked for parallelization.

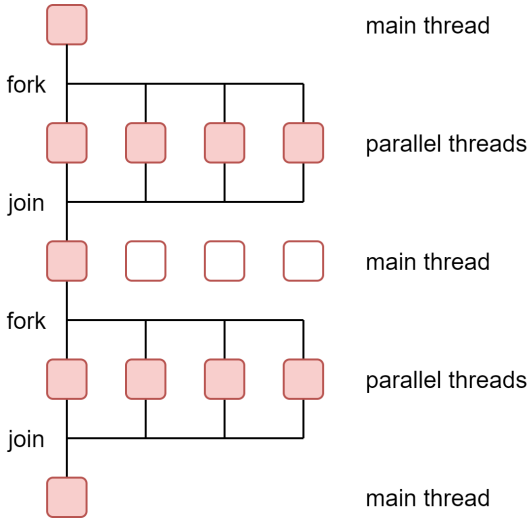


Figure 2.8 OMP fork-join model.

2.3.2 Parallel programming on a DM system

As the clusters (DM-MIMD) grew in size and numbers, a standardized method for efficient communication between distributed resources was needed. Unlike their predecessor, monolithic supercomputers, clusters are significantly less integrated and typically contain several thousands or tens of thousands of distributed processors. Consequently, clusters heavily rely on and are limited by interprocessor communication.

Standards used for messaging in parallel distributed systems are PVM (Parallel Virtual Machine) and MPI (Message Passing Interface). MPI has over the last thirty years supplanted the older PVM standard and expanded upon it significantly. Apart from portability, MPI introduces improvements in communication performance, allows topology specification, etc. Cluster manufacturers typically issue their MPI versions which are optimized for a given architecture and interconnection. Newer versions of MPI standards that are yet to be widely adopted are MPI-2 and recently MPI-3. Transition to newer standards is rather slow as most applications do not utilise any of the newer functions while the dynamic process management introduced in MPI-2 is irrelevant for systems that use batch scheduling.

Open source version Open MPI is also in active development. Large number of MPI versions tailored to specific architectures, interconnections and applications led to a divergence with different implementations excelling in one area and lacking in another. Open MPI is a joint project that aims to merge different MPI versions developed by individual manufacturers and research laboratories into a unified MPI standard which is aware of the architecture and interconnection topology i.e. it includes all manufacturer-built features specific to a given interconnection.

MPI is a standardized and portable messaging system that due to its universality can operate on a large amount of different parallel architectures. The MPI standard defines the syntax and semantics of a core of library routines for users who write parallel codes in Fortran or C/C++. MPI standard provides to users who develops parallel codes the following:

- the ability to send and receive data and messages.
- the ability to create processes on remote processors or computers.

- the ability to monitor the status of a remote processes.
- the ability to send messages and signals to other programs.

Unlike previous parallel models, the programmer using the MPI standard must explicitly define data allocation and communication specifics, which is both a drawback and at the same time an advantage as it allows great flexibility in parallelization. Migration from the sequential code to parallel is complex and requires substantial input. Control over all aspects of the code is explicit. This is certainly an advantage because it allows applications to be used on different parallel computer architectures. MPI model is intended for use on machines with distributed memory, but can be effectively used on systems with shared memory as well.

MPI applications use master-worker process organization. This means that a single, central process, named master, induces and manages all of the worker processes that execute a certain task. Programmers typically start the master MPI process which then induces i.e. activates all the workers and distributes the necessary data. Figure 2.9 shows a master-worker paradigm when four processes are utilised.

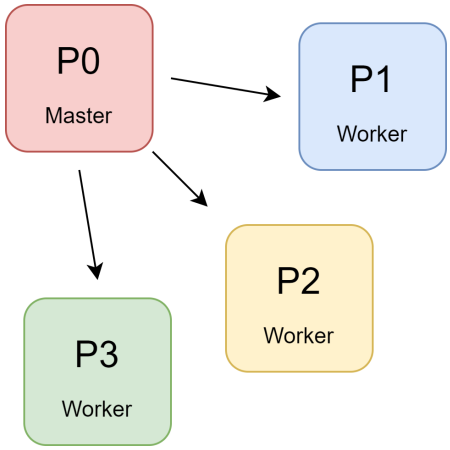


Figure 2.9 Master process P0 initializes workers P1-P3 and distributes data if needed.

After activating all the processes, master process completes his part of the code like all the other workers. If necessary, data can be subsequently exchanged between now equivalent processes according to an explicitly defined communication strategy (Figure 2.10).
Sequential code depicted in 2.1 can be parallelized according to the MPI standard:

Algorithm 2.5 Sequence of code parallelized using MPI standard.

```
1   declare d(n/4),A(n/2,m/2),B(n/2,m/2),C(n/2,m/2)
2   do i=1,n/2
3   d(i) = ... → get d
4   do j=1,m/2
5   A(i,j) = ... → get A, B, C
6   B(i,j) = ...
7   C(i,j) = ...
8   end do
9   end do
10  Call MPI_Send(...)
11  Call MPI_Recv(...)
```

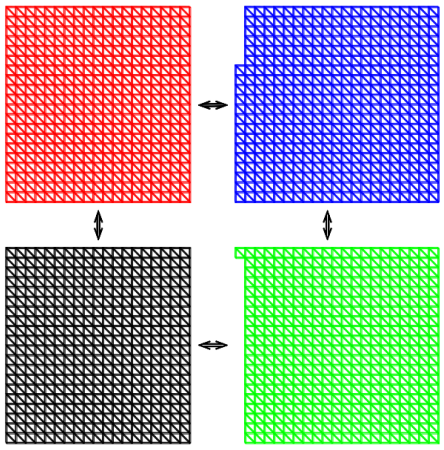


Figure 2.10 Communication is explicitly defined by the programmer when using MPI standard.

2.3.3 Parallel programming using the data distribution strategy

Code to be parallelized using the data distribution approach relies on programmers to distribute the data among processors. Typical data to be distributed are vectors and matrices. High Performance Fortran (HPF) is the software standard governing this type of parallelization. As certain parts of this methodology are difficult to implement, it was often omitted from compilers and has since in most cases been replaced by the OMP-based parallel processing. Code snippet 2.6 depicts HPF parallelization approach.

Algorithm 2.6 Data distribution using HPF.

```
1   !HPF$ PROCESSORS pr(4) → number of processors
2   declare d(n),A(n,m),B(n,m),C(n,m)
3   !HPF$ DISTRIBUTE A(block,block),B(block,block),C(block,block)
4   !HPF$ DISTRIBUTE d(block) → data distribution
5   do i=1,n
6   d(i)=... → get d
7   do j=1,m
8   A(i,j)=... → get A, B, C
9   B(i,j)=...
10  C(i,j)=...
11  end do
12  end do
```

In HPF, code parallelization and communication are governed by the compiler. Successful implementation hence is directly linked to proper data distribution. Fundamental question is how to distribute the data. Nonoptimal distribution will result in intensive communication between the processors, which will be forced to access the necessary data from the memory of other processors, hence the overall program will be several magnitudes slower compared to the sequential code. An obvious solution is data replication, especially if there is significant overlap in data.

Proper data distribution is of primary importance in HPF, but unsuccessful implementation will only affect the overall performance and not the accuracy of the results, since they are not affected by the distribution (unlike OMP). Strategy is extremely efficient, yet restrictive, as only certain algorithms can utilise it. HPF scales well and can be used on both SIMD and MIMD architectures.

2.4 OpenMP and MPI

Italian National Institute for Astrophysics

Partial differential equations play an important role in many branches of science and engineering. The Poisson problem is a partial differential equation (PDE) that is widely applied in many applications. We use this problem as a means for describing the features of MPI and OpenMP that can be used in solving PDEs. We introduce the MPI notion of *virtual topology*, which allows us to manage a grid of processes, describing many of the variations of the basic *send* and *receive* operations supported by MPI.

2.4.1 The Poisson problem

The Poisson problem in two space dimensions x and y takes the form:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) \qquad \text{in the interior} \tag{2.16}$$

$$u(x,y) = g(x,y) \qquad \text{on the boundary} \tag{2.17}$$

An approximate solution can be found using a square *mesh* (also called a *grid*) consisting of points (x_i, y_i), given by

$$x_i = \frac{i}{n+1}\, i = 0,\ldots,n+1,$$

$$y_j = \frac{j}{n+1}\, j = 0,\ldots,n+1$$

with $n+2$ points along each edge of the mesh. The approximate solution $u_{i,j}$ is found only at the points (x_i, y_i). Using the mesh the Eq 2.17 can be rewritten as

$$\frac{u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - 4u_{i,j}}{h^2} = f_{i,j} \tag{2.18}$$

where $h = 1/(n+1)$. Our aim is to solve the latest equation for $u_{i,j}$ everywhere on the mesh. One approach is to rewrite the Eq. 2.18 as

$$u_{i,j} = \frac{1}{4}\left(u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - h^2 f_{i,j}\right) \tag{2.19}$$

iterate by choosing values for all mesh points $u_{i,j}$ and then replace them by using

$$u^{k+1}_{i,j} = \frac{1}{4}\left(u^k_{i-1,j} + u^k_{i,j+1} + u^k_{i,j-1} + u^k_{i+1,j} - h^2 f_{i,j}\right). \tag{2.20}$$

This convergent method is known as *Jacobi iteration*. C code for this is given below.

C Code 2.1 Core algorithm of the Jacobi iteration using C.

```
1 double u[n+2][n+2], unew[n+2][n+2];
2
3 for (int i=1 ; i<=n ; i++)
4         for (int j=1 ; j<=n ; j++)
5                 unew[i,j] = 0.25 * (u[i-1][j] + u[i][j+1] + u[i][j-1]
6                                   + u[i+1][j] - h * h * f[i][j]);
```

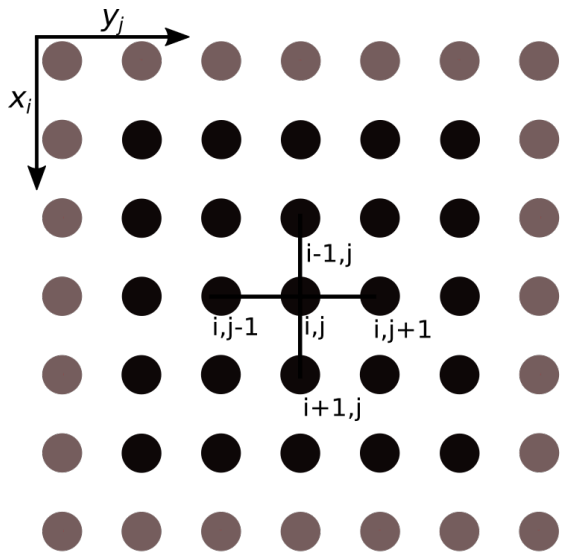


Figure 2.11 Approximation for 2-D Poisson problem, with $n = 5$. The boundaries of the domain are shown in gray.

2.4.2 MPI implementation

To parallelize this algorithm using MPI, it is required to parallelize the loops of the iterations distributing the data, namely the arrays u, $unew$, and f, across the MPI processes. The task is deciding how to assign MPI processes to each part of the decomposed domain. Several approaches are possible for domain decomposition, defining the *application topology* or *virtual topology*. The discussion about the best way for application topology to be fitted onto the physical topology of the parallel computer is beyond the scope of this Chapter.

Jacobi with 1-D decomposition

The simplest decomposition is shown in Figure 2.12, where the physical domain is sliced into slabs along the vertical direction (i.e. 1-D domain decomposition), while the arrays u, $unew$, and f are replicated across the MPI processes. This approach is not the most efficient in terms of memory usage, because the arrays are replicated across MPI process instead of to be distributed. The piece of code 2.2 provides such a domain decomposition into slabs:

C Code 2.2 1-D domain decomposition code.

```
1  ...
2  int rank, NTasks;
3  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4  MPI_Comm_size(MPI_COMM_WORLD, &NTasks);
5
6  /* get the reminder i.e. take into account uneven
7     decomposition of points among the processes */
8  const int rem = (n + 2) % NTasks;
9
10 /* get the amount of data for each MPI process
11    - chunk is supposed to be >= 1 */
12 const int chunk = (n + 2 - rem) / NTasks;
13
14 /* get the slab dimension along vertical direction */
15 int incr, offset;
16 if (rank < rem)
```

```
17    {
18       incr   = chunck + 1;
19       offset = 0;
20    }
21    else
22    {
23       incr   = chunck;
24       offset = rem;
25    }
26 const int start = ((rank * incr) + offset);
27 const int end   = (start + incr);
28
29 /* MPI rank handles the subdomain with the following ranges:
30    -[start-1, end+1]  rows (x_i points, including ghost points).
31    -[0, n+1] columns (y_i points);
32
33 function to handle communications is missing
34
35 Core calculation
36 for (int i=start ; i<=end ; i++)
37         for (int j=1 ; j<=n ; j++)
38                 unew[i,j] = 0.25 * (u[i-1][j] + u[i][j+1] + u[i][j-1]
39                                   + u[i+1][j] - h * h * f[i][j]); */
40 ...
```

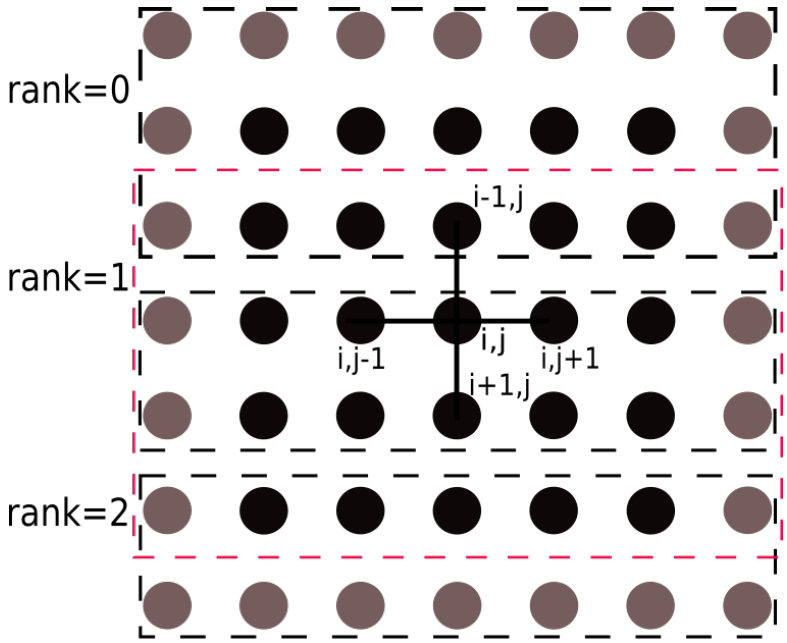


Figure 2.12 1-D domain decomposition, with $n = 5$, across three MPI processes. Black dashed boxes show the computational domain of each MPI process. Red dashed box shows the computational domain, with ghost points, for the MPI process with $rank = 1$.

Looking at the Figure 2.12, the calculation of the element $u[i][j]$, handled by the MPI process with $rank = 1$, will require the element $u[i-1][j]$ that belongs to the MPI process with $rank = 0$. This will imply that data must be exchanged between neighboring MPI processes, and consequently, the array managed by each MPI process must be expanded to hold data from other processes. The elements of the array that are used for MPI communications are called *ghost points*.

With such a domain decomposition, each process with $rank = m$ sends data to the

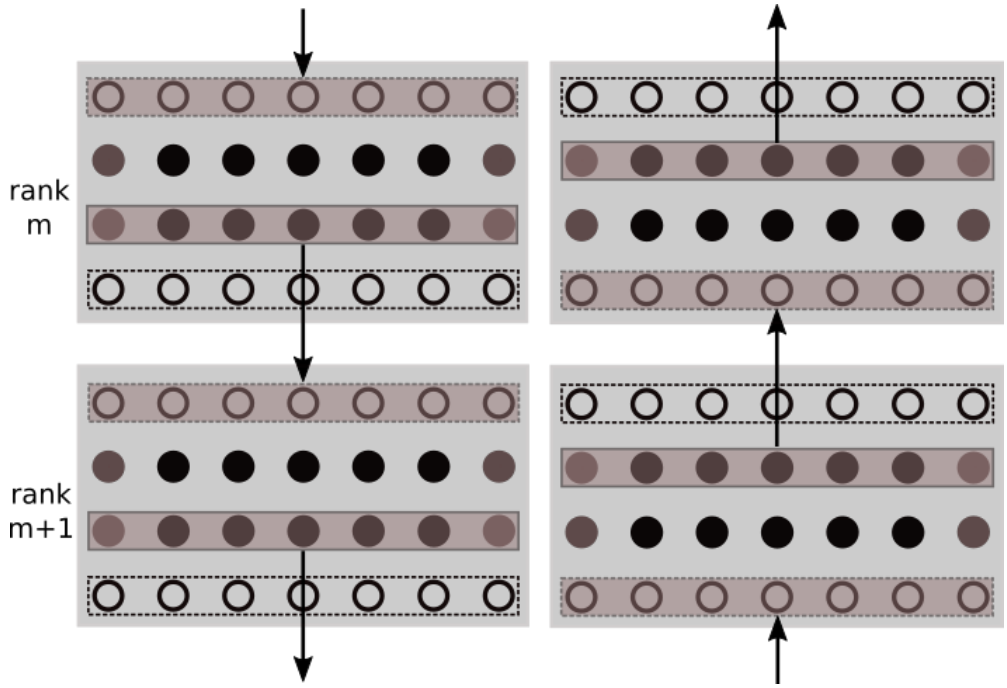


Figure 2.13 Two-step MPI communication between contiguous processes (i.e. $rank = m$ and $rank = m + 1$). Data to be communicated is shaded. The mesh points are shown as black circles, while the ghost points are shown as unfilled circles. During the first stage (on the left), the process with $rank = m$ sends data to the process with $rank = m + 1$ and receives data from the process with $rank = m - 1$ (not shown on the sketch). During the second stage (on the right), the process with $rank = m$ sends data to the process with $rank = m - 1$ (not shown on the sketch) and receives data from the process with $rank = m + 1$.

process with $rank = m + 1$ and then receives data from the process with $rank = m - 1$, as sketched in the left part of the Figure 2.13. After that, the order is reversed, so data is sent to the process with $rank = m - 1$ and received from the process with $rank = m + 1$, as sketched in the right part of the Figure 2.13. The routine 2.3 that accomplishes this task in the following:

C Code 2.3 Code to manage data exchange in 1-D for ghost points using blocking MPI send and receive routines.

```
1  void mpi_exchange_1d(double *const buffer,
2                       const int          n,
3                       const int          nbrtop,
4                       const int          nbrbottom,
5                       const int          start,
6                       const int          end,
7                       const MPI_Comm     comm1d)
8
9  {
10   /* The function is called by each MPI rank
11      - nbrtop    is the MPI process with rank + 1
12      - nbrbottom is the MPI process with rank - 1 */
13
14   /*************** First communication stage *****************/
15   // Perform a blocking send to the top (rank+1) process
16   MPI_Send(&buffer[end][1], n, MPI_DOUBLE_PRECISION,
17            nbrtop, 0, comm1d);
18
19   // Perform a blocking receive from the bottom (rank-1) process
20   MPI_Recv(&buffer[start-1][1], n, MPI_DOUBLE_PRECISION,
```

```
21              nbrbottom, 0, comm1d, MPI_STATUS_IGNORE);
22    /*********************************************************/
23
24    /************** Second communication stage ***************/
25    // Perform a blocking send to the bottom (rank-1) process
26    MPI_Send(&buffer[start][1], n, MPI_DOUBLE_PRECISION,
27              nbrbottom, 1, comm1d);
28
29    // Perform a blocking receive from the top (rank+1) process
30    MPI_Recv(&buffer[end+1][1], n, MPI_DOUBLE_PRECISION,
31              nbrtop, 1, comm1d, MPI_STATUS_IGNORE);
32    /*********************************************************/
33  }
```

Looking at the routine, it should be noted that both processes with the first (i.e. $rank = 0$) and the last rank (i.e. $rank = NTasks - 1$) do not have neighbor processes on the bottom and the top, respectively. This is due to the fact that the grid of the Poisson problem is not periodic. In the MPI implementation, this is indicated by the value **MPI_PROC_NULL**. This value is a valid destination for the **MPI_Send** routine and a valid source for the **MPI_Recv** routine. Passing such a value to one of the above routines is identical to the code of the following form:

C Code 2.4 Snippet of code to show the behaviour of the special value MPI_PROC_NULL.

```
1  ...
2  if (dest != MPI_PROC_NULL)
3   MPI_Send(..., dest, ...);
4  ....
5  if (source != MPI_PROC_NULL)
6   MPI_Recv(..., source, ...);
7  ...
```

The **MPI_PROC_NULL** simplifies the code that is needed for dealing with boundaries.

Although the communication pattern adopted in the *mpi_exchange_1d* routine is frequently used in many applications, it is not the best one because the communication is entirely sequential. The behavior of the communication pattern is shown in Figure 2.14, considering four MPI processes. The sends do not complete until the matching receives take place on the destination process. Since the last process with $rank = 3$ does not perform the (first) send in the *mpi_exchange_1d* routine (i.e the destination process is **MPI_PROC_NULL**), it can immediately receive data from the process with $rank = 2$, thus allowing that process to complete the **MPI_Send** and then to receive data from the process with $rank = 1$ through the **MPI_Recv**, and so forth.

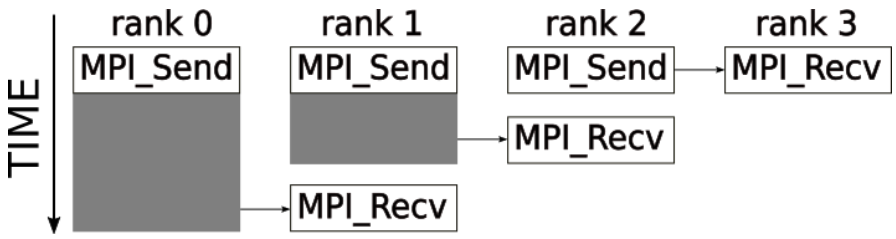


Figure 2.14 MPI sequential communication due to sends blocking until the matching receive is posted. The shaded area shows the idle time, while the process is waiting until the send communication is allowed to transfer the data to the neighboring process.

It is worth understanding in more detail what happens when MPI sends a message. Consider the following code:

C Code 2.5 Snippet of code to show the use of MPI_Send and MPI_Recv routines.

```
...
int err;
if (rank == 0)
    err = MPI_Send(send_buffer, ..., 1, ...);
else if (rank == 1)
    err = MPI_Recv(recv_buffer, ..., 0, ...);
...
```

What happens when the *send_buffer* is sent by the process zero (i.e. with $rank = 0$) but process one (i.e. with $rank = 1$) is not ready to receive it? Three possibilities exist:

1. process zero stops and waits until the process one is ready to receive the message;
2. process zero copies the *send_buffer* to some internal buffer (managed transparently by MPI) and returns successfully from the **MPI_Send** call (i.e. the MPI routine returns the value **MPI_SUCCESS**). For large applications that are already using a large amount of memory, the space available for message buffering may be quite small, so process zero has no choice but to wait for process one;
3. **MPI_Send** routine fails. Note that MPI does not guarantee that an MPI program can continue past an error, however, MPI implementations will attempt to continue whenever possible.

The programmer should be aware of the pitfalls of the MPI implementation, keeping in mind that an MPI implementation is allowed to buffer the message to be sent into internal storage yielding a non-blocking **MPI_Send**, but it is not required to do so. Indeed, the proposed implementation of the *mpi_exchange_1d* routine shows a performance problem, because the code works but it does not execute in parallel.

MPI topology with ordered send and receive

One of the easiest ways to solve the issue is to pair the sends and receives calls. That is, if one process is sending a message to another, the destination will issue a receive call that matches the send before doing in turn a send call, as shown in the routine 2.6. In this routine, the even ranks send first, and the odd ranks receive first, yielding a paired exchange.

C Code 2.6 Code to manage data exchange in 1-D for ghost points using paired MPI sends and receives.

```
void mpi_exchange_paired_1d(double *const buffer,
                            const int          n,
                            const int          nbrtop,
                            const int          nbrbottom,
                            const int          start,
                            const int          end,
                            const MPI_Comm     comm1d)

{
  /* The function is called by each MPI rank
     - nbrtop    is the MPI process with rank + 1
     - nbrbottom is the MPI process with rank - 1 */

  if ((rank % 2) == 0) /* even rank */
  {
    /*************** First communication stage *****************/
    // Perform a blocking send to the top (rank+1) process
    MPI_Send(&buffer[end][1], n, MPI_DOUBLE_PRECISION,
             nbrtop, 0, comm1d);

    // Perform a blocking receive from the bottom (rank-1) process
    MPI_Recv(&buffer[start-1][1], n, MPI_DOUBLE_PRECISION,
```

```
            nbrbottom, 0, comm1d, MPI_STATUS_IGNORE);
    /*********************************************************/

    /************** Second communication stage ****************/
    // Perform a blocking send to the bottom (rank-1) process
    MPI_Send(&buffer[start][1], n, MPI_DOUBLE_PRECISION,
            nbrbottom, 1, comm1d);

    // Perform a blocking receive from the top (rank+1) process
    MPI_Recv(&buffer[end+1][1], n, MPI_DOUBLE_PRECISION,
            nbrtop, 1, comm1d, MPI_STATUS_IGNORE);
    /*********************************************************/
  }
  else /* odd rank */
  {
    /************** First communication stage ****************/
    // Perform a blocking receive from the bottom (rank-1) process
    MPI_Recv(&buffer[start-1][1], n, MPI_DOUBLE_PRECISION,
            nbrbottom, 0, comm1d, MPI_STATUS_IGNORE);

    // Perform a blocking send to the top (rank+1) process
    MPI_Send(&buffer[end][1], n, MPI_DOUBLE_PRECISION,
            nbrtop, 0, comm1d);
    /*********************************************************/

    /************** Second communication stage ****************/
    // Perform a blocking receive from the top (rank+1) process
    MPI_Recv(&buffer[end+1][1], n, MPI_DOUBLE_PRECISION,
            nbrtop, 1, comm1d, MPI_STATUS_IGNORE);

    // Perform a blocking send to the bottom (rank-1) process
    MPI_Send(&buffer[start][1], n, MPI_DOUBLE_PRECISION,
            nbrbottom, 1, comm1d);
    /*********************************************************/
  }
}
```

MPI topology with combined send and receive

Pairing MPI communications is effective but can be difficult to program when the MPI topology is complex. A productive alternative is the usage of the **MPI_Sendrecv** routine, which allows both to send and receive data without deadlock. The code for the combined send-receive is shown in the routine 2.7.

C Code 2.7 Code to manage data exchange in 1-D for ghost points using send-receive MPI routine.

```
void mpi_exchange_sendrecv_1d(double *const buffer,
                              const int       n,
                              const int       nbrtop,
                              const int       nbrbottom,
                              const int       start,
                              const int       end,
                              const MPI_Comm  comm1d)

{
/* The function is called by each MPI rank
   - nbrtop    is the MPI process with rank + 1
   - nbrbottom is the MPI process with rank - 1 */

MPI_Sendrecv(
    &buffer[end][1],     n, MPI_DOUBLE_PRECISION, nbrtop,     0,
```

```
16      &buffer[start-1][1], n, MPI_DOUBLE_PRECISION, nbrbottom, 0,
17      comm1d, MPI_STATUS_IGNORE
18      );
19
20  MPI_Sendrecv(
21      &buffer[start][1], n, MPI_DOUBLE_PRECISION, nbrbottom, 1,
22      &buffer[end+1][1], n, MPI_DOUBLE_PRECISION, nbrtop,    1,
23      comm1d, MPI_STATUS_IGNORE
24      );
25  }
```

MPI also allows the programmer to provide a buffer into which the message can be placed until it is delivered through the **MPI_Buffer_attach** routine. Then it is enough to replace the **MPI_Send** call with **MPI_Bsend**.

MPI topology with nonblocking communications

The programmer should always keep in mind that moving data from one process to another takes more time (and usually required more energy) than processing data within a single process (core). This mismatch reflects the underlying hardware capability and should be one of the main concerns when the programmer designs a parallel application that relies on a message-passing approach.

The nonblocking send and receive operations are issued by the **MPI_Isend** and **MPI_Irecv** routines. The arguments of such routines are the same as for **MPI_Send** and **MPI_Recv** with the addition of a *handle* as the last argument (in C language). The general rule is that the buffer containing the message issued through the call of **MPI_Isend** has not to be modified until the message has been received. MPI allows checking for the delivery of the message using the **MPI_Wait** or **MPI_Test** routines, or it provides a way to wait for all the nonblocking operations through the **MPI_Waitall** routine. The code using nonblocking communications is shown in the routine 2.8.

C Code 2.8 Code to manage data exchange in 1-D for ghost points using nonblocking MPI routine.

```
1  void mpi_exchange_nonblocking_1d(double *const buffer,
2                                   const int          n,
3                                   const int          nbrtop,
4                                   const int          nbrbottom,
5                                   const int          start,
6                                   const int          end,
7                                   const MPI_Comm     comm1d)
8
9  {
10   /* The function is called by each MPI rank
11      - nbrtop    is the MPI process with rank + 1
12      - nbrbottom is the MPI process with rank - 1 */
13
14   // communication request (handle)
15   MPI_Request request[4];
16
17   /************** Process is ready to receive data ***************/
18   // Perform a nonblocking receive from the bottom (rank-1) process
19   MPI_Irecv(&buffer[start-1][1], n, MPI_DOUBLE_PRECISION,
20             nbrbottom, 0, comm1d, &request[0]);
21
22   // Perform a nonblocking receive from the top (rank+1) process
23   MPI_Irecv(&buffer[end+1][1], n, MPI_DOUBLE_PRECISION,
24             nbrtop, 1, comm1d, &request[1]);
25   /***************************************************************/
26
```

```
27    /************** Process begins to send data ********************/
28    // Perform a nonblocking send to the top (rank+1) process
29    MPI_Isend(&buffer[end][1], n, MPI_DOUBLE_PRECISION,
30              nbrtop, 0, comm1d, &request[2]);
31
32    // Perform a nonblocking send to the bottom (rank-1) process
33    MPI_Isend(&buffer[start][1], n, MPI_DOUBLE_PRECISION,
34              nbrbottom, 1, comm1d, &request[3]);
35    /*************************************************************/
36
37    /* some useful work (computation) could be performed (while the
38       buffer containing the message has not to be modified) */
39
40    /************ Waits for all given MPI Requests to complete *******/
41    MPI_Waitall(4, request, MPI_STATUSES_IGNORE);
42  }
```

In principle, this routine can be twice as fast as the version 2.6. Note that the main purpose of using nonblocking MPI routines is to overlap communication and computation in programs. This means that after **MPI_Isend** and **MPI_Irecv** have been issued some useful work (computation) could be performed (while the buffer containing the message has not to be modified) before the **MPI_Wait** or **MPI_Waitall** routine is called. The latest routine guarantees that the process waits for all given MPI requests to complete (in our case **MPI_Isend** and **MPI_Irecv** calls).

Jacobi with 2-D decomposition

Until now we show how to numerically tackle the Poisson problem with 1-D decomposition, where the physical domain is sliced into slabs along the vertical direction. Another important virtual topology is the *Cartesian* topology, which is simply a decomposition in the natural coordinate (e.g. x,y) directions. MPI provides a collection of routines for defining and managing the Cartesian topology.

C Code 2.9 MPI Cartesian topology.

```
1  ...
2  // communicator size
3  #define SIZE 2
4  // X direction
5  #define X    0
6  // Y direction
7  #define Y    1
8
9  int ThisTask, NTasks;
10 MPI_Comm comm2d;
11
12 static int dims[SIZE] = {2, 3};
13 static int periods[SIZE] = {0, 0};
14 static int reorder = 0;
15
16 // initialize the MPI execution environment
17 MPI_Init(NULL, NULL);
18
19 // determines the size of the group associated with a communicator
20 MPI_Comm_size(MPI_COMM_WORLD, &NTasks);
21
22 // determines the rank of the calling process in the communicator
23 MPI_Comm_rank(MPI_COMM_WORLD, &ThisTask);
24
25 /* makes a new communicator to which
```

```
26    topology information has been attached */
27 MPI_Cart_create(MPI_COMM_WORLD, SIZE, dims, periods, reorder, &comm2d);
28 ...
29 int coords[SIZE];
30 /* determines process coords in
31    cartesian topology given rank in group */
32 MPI_Cart_coords(comm2d, ThisTask, SIZE, coords);
33 ...
34 int rank_source, rank_dest;
35 /* returns the shifted source and destination ranks,
36    given a shift direction and amount */
37 MPI_Cart_shift(comm2d, X, 1, &rank_source, &rank_dest);
38 ...
```

The routine 2.9 though **MPI_Cart_create** creates a new communicator in the sixth argument (*comm2d*) from the (old) communicator specified in the first argument (*MPI_COMM_WORLD*). The features of the Cartesian topology are assigned by the second through the fifth argument. The *dims* argument specifies the x and y dimensions of the Cartesian mesh. The 1-D decomposition topology discussed in Section 2.4.2 can also be configured using the MPI Cartesian routine setting up to one the dimension along the y axis. The *periods* argument indicates whether the mesh is periodic or not in each dimension. Setting the logical value *reorder* to *TRUE* allows MPI to reorder the rank of the processes figuring out the neighbors in the actual hardware for better performance. At runtime, it is possible to retrieve full Cartesian topology information associated with a communicator using the **MPI_Cart_get** routine. This routine returns i) the number of processes for each cartesian dimension, ii) the periodicity (true/false) for each cartesian dimension, and iii) the coordinates of the calling process in the cartesian structure. Coordinates in Cartesian topology can also be retrieved using the **MPI_Cart_coords** routine, which takes as input the rank of the MPI process.

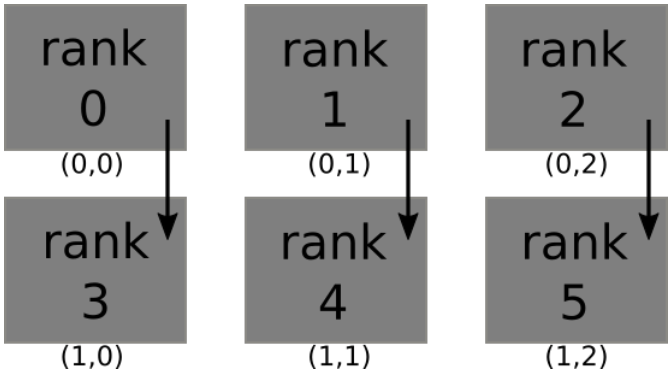


Figure 2.15 A 2-D Cartesian domain decomposition using six MPI processes, also showing a shift by one in the first dimension. This domain decomposition is obtained through the code 2.9.

Figure 2.15 shows the obtained 2-D Cartesian decomposition using the routine 2.9. Following the previous arguments in order to solve the Poisson problem, each MPI process needs to send and receive data from its neighbors, as illustrated in Figure 2.13 (using the 2-D decomposition each process needs to communicate not only with top and bottom processes but also with left and right processes). The **MPI_Cart_shift** routine, as shown in the code 2.9, allows us to figure out the neighboring processes, i.e. it returns the shifted source and destination ranks, given a shift direction and amount. For example, the process with $rank = 1$ at Cartesian coordinates (0,1) in Figure 2.15, along the X (vertical) direction, has *rank_source* = ***MPI_PROC_NULL*** (because the

domain is not periodic) and *rank_dest* = 4.

Using the 2-D Cartesian domain decomposition the MPI communication pattern implemented in the routine 2.8 is no longer valid because now we need to figure out right and left neighbors as well as the top and bottom neighbors, as shown by the following routine:

C Code 2.10 Left, right, top and bottom neighbors in 2-D cartesian domain decomposition.

```
...
/* comm2d is the 2-D Cartesian communicator previously created
   get left and right neighbors (Y direction) */
int nbrright, nbrleft;
MPI_Cart_shift(comm2d, Y, 1, &nbrleft, &nbrright);

// get bottom and top neighbors (X direction)
int nbrtop, nbrbottom;
MPI_Cart_shift(comm2d, X, 1, &nbrbottom, &nbrtop);
...
```

We change the body of the core algorithm of the Jacobi iteration accordingly:

C Code 2.11 Core algorithm of the Jacobi iteration in 2-D Cartesian domain decomposition.

```
...
/* Using 2-D Cartesian domain decomposition each MPI rank handles
   the subdomain with the following ranges:
   - [x_start-1, x_end+1] (x_i points, including ghost points).
   - [y_start-1, y_end+1] (y_i points, including ghost points); */

// New ranges for the core calculation

for (int i=x_start ; i<=x_end ; i++)
    for (int j=y_start ; j<=y_end ; j++)
        unew[i,j] = 0.25 * (u[i-1][j] + u[i][j+1] + u[i][j-1]
                          + u[i+1][j] - h * h * f[i][j]);
 ...
```

The last routine that we need to design is the 2-D version of the data exchange routine 2.8. Data communication with top and bottom neighbors remains the same, while is a little more difficult when it involves the left and right processes because the data is not stored contiguously in memory.

In the data exchange routines seen so far, buffers are contiguous areas in memory, so that, for example, the datatypes *MPI_DOUBLE_PRECISION* coupled with the count of occurrences are sufficient to describe the buffer to be sent. MPI *derived datatypes* allow the programmer to specify *non-contiguous* areas in memory, such as a column of an array stored in a row-major order like in C programming language. The mechanism provided by MPI for describing a new kind of datatype, interesting for our purpose, is the following:

C Code 2.12 MPI derived datatype routine.

```
...
/* Creates a vector (strided) datatype:
   - 1st argument (count)      : number of blocks;
   - 2nd argument (blocklength): number of elements in each block
   - 3rd argument (stride)     : number of elements between the
                                 start of each block
   - 4th argument (oldtype)    : old datatype
   - 5th argument (newtype)    : new datatype

```

```
   count = (x_end - x_start + 1), i.e. the size of each column
   blocklength = 1, i.e. one double (old datatype) in each block
   stride = (y_end - y_start + 3), i.e. ghost points are discarded
   oldtype = MPI_DOUBLE_PRECISION */

MPI_Datatype column;
MPI_Type_vector((x_end - x_start + 1), 1, (y_end - y_start + 3),
                MPI_DOUBLE_PRECISION, &column);

// commits the datatype
MPI_Type_commit(column);
...
/* when the datatype is no longer needed, it should be freed,
  i.e. datatype is set to MPI_TYPE_NULL */
MPI_Type_free(column);
```

Finally, we have all the necessities to implement the two-dimensional exchange routine required when the 2-D Cartesian domain decomposition is used.

C Code 2.13 Code to manage the exchange in 2-D Cartesian domain for ghost points using send-receive MPI routine .

```
void mpi_exchange_sendrecv_2d(double *const buffer,
                              const int          nbrtop,
                              const int          nbrbottom,
                              const int          nbrleft,
                              const int          nbrright
                              const int          x_start,
                              const int          x_end,
                              const int          y_start,
                              const int          y_end,
                              const MPI_Comm     comm2d,
                              const MPI_Datatype column)

{
  const int data_row_size = (y_end - y_start + 1);
  MPI_Sendrecv(&buffer[x_end][y_start],     data_row_size,
               MPI_DOUBLE_PRECISION, nbrtop,    0,
               &buffer[x_start-1][y_start], data_row_size,
               MPI_DOUBLE_PRECISION, nbrbottom, 0,
               comm2d, MPI_STATUS_IGNORE);

  MPI_Sendrecv(&buffer[x_start][y_start], data_row_size,
               MPI_DOUBLE_PRECISION, nbrbottom, 1,
               &buffer[x_end+1][y_start], dat_row_size,
               MPI_DOUBLE_PRECISION, nbrtop,    1,
               comm2d, MPI_STATUS_IGNORE);

  MPI_Sendrecv(&buffer[x_start][y_end],     1, column, nbrright, 0,
               &buffer[x_start][y_start-1], 1, column, nbrleft,  0,
               comm2d, MPI_STATUS_IGNORE);

  MPI_Sendrecv(&buffer[x_start][y_start], 1, column, nbrleft,  1,
               &buffer[x_start][y_end+1], 1, column, nbrright, 1,
               comm2d, MPI_STATUS_IGNORE);
}
```

Computation and communication overlapping

In real applications programmers figure out an effective technique for masking data transfer latency, with the potential for considerable performance gains, enabling applications to scale well on a large number of processing units. So far, we design

the communication pattern using nonblocking routines in order to avoid deadlock in the communications, but we do not discuss how to arrange the program so that some useful work can be done while processes are performing communications.

Looking at the code 2.1, in the core algorithm of the Jacobi iteration the values of the buffer *unew* at points of the mesh that are interior to the domain (i.e. values $(x\_start+1) \leq x_i \leq (x\_end-1)$, and $(y\_start+1) \leq y_i \leq (y\_end-1)$) on each process can be computed without data exchange with the other processes. We call these values *local data*.

Hence, the computation and communication can be arranged in such a way:

(I) begin nonblocking sending/receiving data to/from the other processes;
(II) perform the computation with the local data;
(III) check if data have been received from the other processes and finish computing with them.

The snippet of code, assuming the 2-D Cartesian domain decomposition, for the task (I) is the following:

C Code 2.14 Snippet of code to begin nonblocking send/recv to/from the other processes.

```
...
/* 2-D Cartesian domain decomposition is assumed
  4 nonblocking MPI_Irecv + 4 nonblocking MPI_Isend */
MPI_Request request[8];

MPI_Irecv(..., nbrbottom, 0, comm2d, &request[0]);
MPI_Irecv(..., nbrtop,    1, comm2d, &request[1]);
MPI_Irecv(..., nbrleft,   2, comm2d, &request[2]);
MPI_Irecv(..., nbrright,  3, comm2d, &request[3]);

MPI_Isend(..., nbrtop,    0, comm2d, &request[4]);
MPI_Isend(..., nbrbottom, 1, comm2d, &request[5]);
MPI_Isend(..., nbrright,  2, comm2d, &request[6]);
MPI_Isend(..., nbrleft,   3, comm2d, &request[7]);
...
```

The computation routine for the task (II) is the following:

C Code 2.15 Core algorithm of the Jacobi iteration assuming the 2-D Cartesian domain decomposition on local data.

```
for (int i=x_start+1 ; i<x_end ; i++)
    for (int j=y_start+1 ; j<y_end ; j++)
        unew[i,j] = 0.25 * (u[i-1][j] + u[i][j+1] + u[i][j-1]
                          + u[i+1][j] - h * h * f[i][j]);
```

Finally, the snippet of code for the task (III) is the following:

C Code 2.16 Snippet of code to finalize the calculation on points of the grid that require ghost points.

```
...
int idx;
MPI_Status status;

for (int req=0 ; req<8 ; req++)
{
    /* waits for any specified MPI Request to complete */
    MPI_Waitany(8, request, &idx, &status);

    switch(status.MPI_TAG)
    {
        /* commuication with tag 0 completed */
        case 0:
```

```
            for (int j=y_start ; j<=y_end ; j++)
                unew[x_start][j] = ...
            break;

        /* communication with tag 1 completed */
        case 1:
            for (int j=y_start ; j<=y_end ; j++)
                unew[x_end][j] = ...
            break;

        /* communication with tag 2 completed */
        case 2:
            for (int i=x_start ; i<=x_end ; i++)
                unew[i][y_start] = ...
            break;

        /* communication with tag 3 completed */
        case 3:
            for (int i=x_start ; i<=x_end ; i++)
                unew[i][y_end] = ...
            break;
    }
}
...
```

2.4.3 OpenMP implementation

The Jacobi algorithm with a serial code has:

- time complexity of the order $O(I \cdot N^2)$, where I is the number of iterations to achieve convergence (usually the iteration is terminated when the difference between two successive approximations to the solution is less than $\sim 10^{-5}$) and N is the grid size (i.e. nested loops of the core algorithm of the Jacobi iteration 2.1);
- memory complexity of the order $O(N^2)$.

Consequently, we expect:

- long times to process matrix of big size and/or great amount of iterations (cache misses rate severely impacts performance when the matrix size exceeds cache size);
- size of the matrix is limited by the platform (i.e. size of the memory available on the node). On modern platforms, this may be a minor issue.

We remind that the main loop of the Jacobi iterator takes the form:

C Code 2.17 The main loop of the Jacobi iteration.

```
#define MAX_COUNT 10000
#define TOLERANCE 1.0e-5

int iCount = 1;
double err = 1.0;
while((iCount < MAX_COUNT) && (err > TOLERANCE))
{
    for (int i=0 ; i<Dimension ; i++)
        for (int j=0 ; j<Dimension ; j++)
            {...}

    iCount++;
```

```
13      err = DIFF(...);
14  }
```

The while loop cannot be parallelized using explicitly the *#pragma omp parallel for* directive, because the iterations of the while loop are not specified. However, the inner loops can be easily parallelized, as shown in the following code:

C Code 2.18 The first inner loop of the Jacobi iteration parallelized using *OpenMP*.

```
1  while((iCount < MAX_COUNT) && (err > TOLERANCE))
2  {
3    #pragma omp parallel default(none) private(i,j)
4                                       shrared (Dimensions, {...})
5                                       num_threads(NThreads)
6    {
7      #pragma omp for schedule(static)
8      for (int i=0 ; i<Dimension ; i++)
9          for (int j=0 ; j<Dimension ; j++)
10             {...}
11   } /* omp parallel region */
12
13   iCount++;
14   err = DIFF(...);
15 } /* while loop */
```

Using the clause *schedule(static)* on the *omp for* directive, the loop over the *i* index is parallelized across the threads with chunk size equal to *Dimension/NThredas*, where *NThreads* is the number of OpenMP threads working within the parallel region, specified through the clause *num_threads(integer-expression)*. With most OpenMP runtimes, the default scheduling when no schedule clause is specified is *static*. However, to provide the maximum application flexibility (i.e. do not modify the source code of the application), it is possible to set the maximum number of threads to use for OpenMP parallel regions using the **OMP_NUM_THREADS** environment variable, and to set the run-time schedule type and optional chunk size through the **OMP_SCHEDULE** environment variable, while *static* and *num_threads* clauses are not specified in the source code.

Looking carefully at the loops in the source code 2.18, we figure out that the loops are (i) perfectly nested, i.e. there is no intervening code between the loops, (ii) they form a rectangular iteration space and the bounds of each loop are invariant over all the loops, (iii) the instructions of the innermost associated loop do not contain any *break* statement nor *continue* statement. This means that the two loops can be parallelized in a nest without introducing nested parallelism. The latest task is accomplished through the *omp collapse* clause, as shown in the following snippet of code:

C Code 2.19 Usage of the *collapse* clause on the *omp for* loop to parallelize multiple loops in a nest.

```
1  ...
2    {
3      #pragma omp for collapse(2) schedule(static)
4      for (int i=0 ; i<Dimension ; i++)
5          for (int j=0 ; j<Dimension ; j++)
6              {...}
7    } /* omp parallel region */
```

The resulting collapsed loop is parallelized across the threads with chunk size equal to $Dimension^2/NThreads$.

We point out that, using this approach, the parallel section is created and destroyed (fork-join model) for every iteration. The master thread runs from start to finish the

program. When the parallel region is encountered, additional threads are created by the runtime system. The master thread is included in the group of active threads within the parallel region. At the termination of the parallel region, all the threads are synchronized, and the execution continues only after the last thread has arrived at the implicit barrier. After the parallel region exits, the master thread continues updating the value of the counter *iCount*, evaluating the error *err* and the logical value of the *while* loop. This approach might lead to a performance penalty because the threads are forked and joined in each iteration of the *while* loop. However, "smart" OpenMP implementations do not join threads after a parallel region is completed; instead, threads might busy-wait for some time, and then sleep until another parallel region is started. This is done exactly to prevent high overheads at the start and the end of parallel regions within a loop, so the impact on the execution time might be negligible. The **OMP_WAIT_POLICY** environment variable provides hints about the preferred behavior of waiting threads during program execution. Use *ACTIVE* if you want waiting threads to mostly be active. That is, the threads consume processor cycles while waiting. This wait policy is recommended for maximum performance on the dedicated machine.

With the introduction of the *cancellation constructs* in OpenMP 4.0, an elegant way to terminate the execution of a *parallel* construct is available. This feature is useful for specific parallel methods with dynamic behavior, such as, for instance, the *while* work-sharing loop. The *cancel* and *cancellation point* constructs in C/C++ are stand-alone directives, which trigger an action when, or if, is encountered at runtime by any thread in its execution path.

An elegant and more sophisticated way to support unstructured parallelism, such as unbound loops and recursive functions, is offered by the tasking execution model (not covered here), first introduced in OpenMP 3.0, and refined in later versions.

2.5 GPU computing

University of Rijeka

In this section we will focus on CUDA (Compute Unified Device Architecture), a computing platform that facilitates programming on graphics cards produced by Nvidia and OpenACC, a programming standard for accelerators.

2.5.1 CUDA

The term CUDA refers both to software and underlying hardware architecture. As a programming language, CUDA is a high-level C/C++ like language intended for heterogeneous programming. It is actively developed, includes many "extensions" and can be accessed/modified from other languages e.g. python (pyCuda). On a hardware level, CUDA-enabled GPUs are massively parallelized computer architectures with shared memory i.e. SIMD machines.

A CUDA program uses kernels which execute certain operations/instructions on the data stream. The kernel is a C++ function callable from the host computer and executed n times in parallel on the CUDA device by n different CUDA threads. A typical example of a data stream is a vector containing n floats.

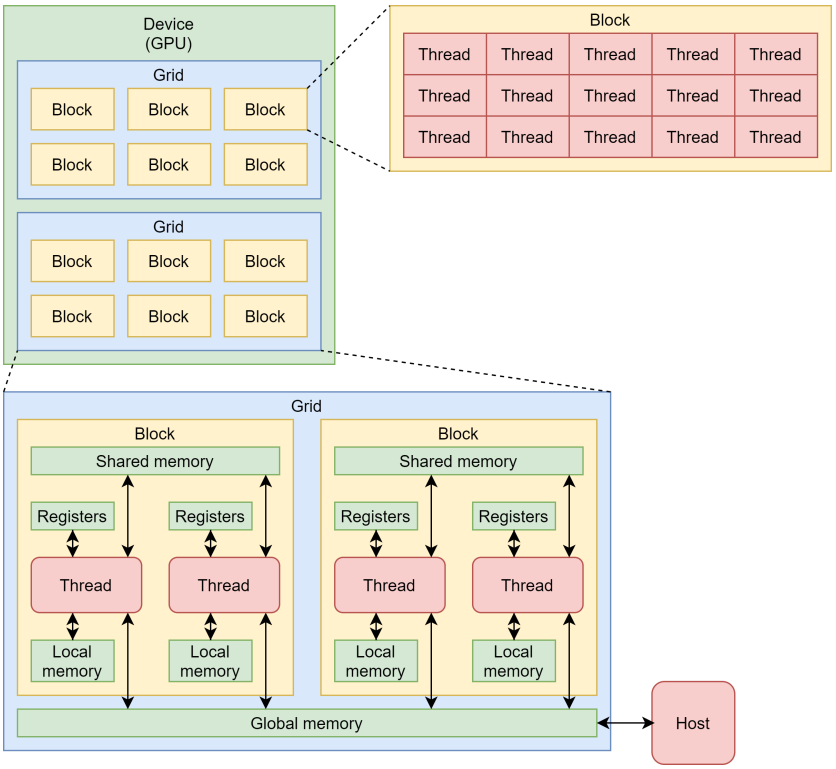


Figure 2.16 CUDA hierarchy.

Memory and threads inside a GPU follow an organizational hierarchy as seen in figure 2.16. Processing elements (threads) are organized as follows:

1. thread.
2. block composed of several concurrently executing threads.
3. grid composed of several concurrently executing thread blocks.

whereas memory organization coincides with the organization of processing elements:

1. per-thread local memory visible to a thread.
2. per-block shared memory visible to all threads in a block.
3. per-device global memory.

Every thread in a device has its own registers and off-chip local memory. All threads in a block can further access a shared memory private to that block and global, per-device memory. This means that, aside from thread-specific memory, there are several levels of shared memory each innate to an organizational category i.e. grids and blocks. The main advantages of CUDA are:

- massive acceleration for parallelizable problems.
- rapidly expanding third-party libraries for machine learning, linear algebra, etc.

The main disadvantages include:

- runs only on Nvidia GPUs.
- high initial cost.
- can be complex to program and requires knowledge of architecture for best efficiency.
- communication overhead between CPU and GPU.
- small community.

CUDA program

We will analyse CUDA imoplementation on a simple vector addition program which evaluates:

$$\mathbf{a} + \mathbf{b} = \mathbf{c} \tag{2.21}$$

First, we need to define the number of elements in an array (vector).

CUDA Code 2.1 Defining a vector.

```
1        int n = 100000;
```

Next, arrays on the host (CPU) and device (GPU) need to be defined. Prefix *d* is used to distinguish the device arrays.

CUDA Code 2.2 Defining arrays on the CPU and GPU.

```
1        double *a;
2        double *b;
3        double *c;
4
5        double *d_a;
6        double *d_b;
7        double *d_c;
```

Since all arrays are double precision, memory required for an array with n elements equals:

CUDA Code 2.3 Calculating memory requirements.

```
1        size_t bytes = n*sizeof(double);
```

Now we need to allocate the memory on the host (CPU) and device (GPU) for arrays **a**, **b**, **c** and **d_a**, **d_b**, **d_c**. *malloc* allocates the requested memory on the host and returns a pointer. *cudaMalloc* allocates the requested memory on the device and returns a pointer. Note the differences in syntax.

CUDA Code 2.4 Allocating memory on the CPU and GPU.

```
1        a = (double*)malloc(bytes);
2        b = (double*)malloc(bytes);
3        c = (double*)malloc(bytes);
4
5        cudaMalloc(&d_a, bytes);
6        cudaMalloc(&d_b, bytes);
7        cudaMalloc(&d_c, bytes);
```

Let's fill arrays **a** and **b** with some values. We must iterate through the elements:

CUDA Code 2.5 Generating data.

```
1        for( i = 0; i < n; i++ ){
2                a[i] = 1.0;
3                b[i] = 4.0;
4        }
```

The next logical step is the transfer of data from the host to the device. This is achieved with *cudaMemcpy* function which takes destination and source arrays, data type and type of transfer:

CUDA Code 2.6 Data transfer from the host to the device.

```
1        cudaMemcpy(d_a, a, bytes, cudaMemcpyHostToDevice);
2        cudaMemcpy(d_b, b, bytes, cudaMemcpyHostToDevice);
```

Now we must set the execution configuration parameters. Threads that can execute an instruction are grouped in blocks of a given size. Each thread has a unique index which has to be defined. Blocks can contain up to 1024 threads, but this number may vary depending on the GPU architecture and CUDA version. Threads in a block share memory. Let's assume we have 256 threads in a block:

CUDA Code 2.7 Number of threads per block.

```
1        int thr_per_blk = 256;
```

Multiple blocks are grouped in a grid. Each block in a grid must have the same number of threads. Block are independent i.e. no communication is possible between them hence this must be accounted for when programming. Grids can be one or two-dimensional. Based on a set number of elements in an array and the number of threads in a block, we can calculate the number of blocks we need to complete the calculation:

CUDA Code 2.8 Calculating the required number of blocks.

```
1        int blk_in_grid = ceil( float(n) / thr_per_blk );
```

The kernel function is launched with:

CUDA Code 2.9 Launching the CUDA kernel.

```
1        vAdd<<< blk_in_grid, thr_per_blk >>>(d_a, d_b, d_c, n);
```

We will return to the kernel and its definition later.

Results of the computation are stored in array **d_c** on the GPU. This data must be copied back to the host array **c**. We will use the same *cudaMemcpy* function, albeit with different source/destination and transfer type:

CUDA Code 2.10 Data transfer from the device to the host.

```
1        cudaMemcpy(c, d_c, bytes, cudaMemcpyDeviceToHost);
```

Let's verify the results. We will sum all the values in **c** and divide them with the number of elements. The result should be 5.0:

CUDA Code 2.11 Results validation.

```
1         double sum = 0;
2         for( i = 0; i < n; i++ )
3         sum += d_c[i];
4         printf("Result: %f\n", sum/n);
```

Finally, we can free the memory on the device and host. To achieve this we will use *cudaFree* and *free* commands:

CUDA Code 2.12 Memory release.

```
1         cudaFree(d_a);
2         cudaFree(d_b);
3         cudaFree(d_c);
4
5         free(a);
6         free(b);
7         free(c);
```

Let's focus on the kernel. CUDA kernel is a "global" function, meaning it is called from the host (CPU) and executed on the device (GPU). It includes the id variable which assigns a unique thread ID to all threads in the grid. The ID of a thread is calculated according to:

CUDA Code 2.13 Defining the thread IDs.

```
1         int id = blockDim.x * blockIdx.x + threadIdx.x;
```

We have previously mentioned that our blocks have 256 threads i.e. $blockDim.x = 256$. As we iterate through blocks, the ID of a given block will assume values of $0,1...N$. Consequently, the ID of a "first" thread in a block will be $0,256,512...n$. Each thread must calculate the sum:

CUDA Code 2.14 Thread iterator.

```
1         if ( id < n )
2         c[id] = a[id] + b[id];
```

If statement ensures that operations are executed only on elements for which we have allocated the memory. This is necessary since we could have reserved more threads than are needed but haven't assigned data/allocated memory.

CUDA programs must first be compiled. On HPC Bura, the CUDA environment is loaded with:

CUDA Code 2.15 Setting up the evironment using modules.

```
1         module load nvidia-CUDA/10.2
```

To build a binary (executable) one must run the following command, assuming that the code is stored in **vectorAdd.cu**:

CUDA Code 2.16 Compilation.

```
1         nvcc vectorAdd.cu -o vectorAddBinary
```

The reated binary is executed with:

CUDA Code 2.17 Code execution

```
1         ./vectorAddBinary
```

Part II

Executing programs and code in HPC environment

1. Workload managers

Introduction
SLURM
PBS
Alternative solutions

1.1 Introduction

University of Rijeka

HPC systems are comprised of multiple computational nodes which form a cohesive unit. Users accessing HPC systems typically access these computational resources through a login node where resources needed for tasks that are to be executed are reserved / requested and subsequently assigned by a workload manager. In other terms, users can not access the computational segment of an HPC system and interact mainly with a specific piece of software that is designed to manage such a system.

Workload managers, resource managers and job schedulers are control systems that track and manage the use of computational resources in an HPC system. Amongst others, they:

- manage resource distribution
- manage job queueing
- assign priorities
- allow job control
- provide insight into historical resource use / allocation.

Common workload managers in HPC systems nowadays are:

- SLURM
- PBS
- LSF.

In this chapter, we will briefly cover more prominent open-source workload managers.

1.2 SLURM

University of Rijeka

Simple Linux Utility for Resource Management or SLURM [SchedMD, 2021] is one of the most widespread workload managers on HPC systems. Its adoption can be attributed to its overall simplicity and open-source nature. Although it might not be as comprehensive of a solution as its competitors, nor is it a meta-batch system, SLURM integrates several useful tools and can be extended with custom plugins.

SLURM utilizes a central **slurmctld** daemon running on a management node with distributed **slurmd** daemons running on compute nodes. **slurmctld** conducts management tasks whereas **slurmd** is responsible for the task / job execution. These daemons are mandatory and enable fault-tolerant communication. Additionally, **slurmdbd** daemon can be optionally employed to monitor and log the accounting information [SchedMD, 2021]. The hierarchical structure and main components of SLURM are shown in Figure 1.1.

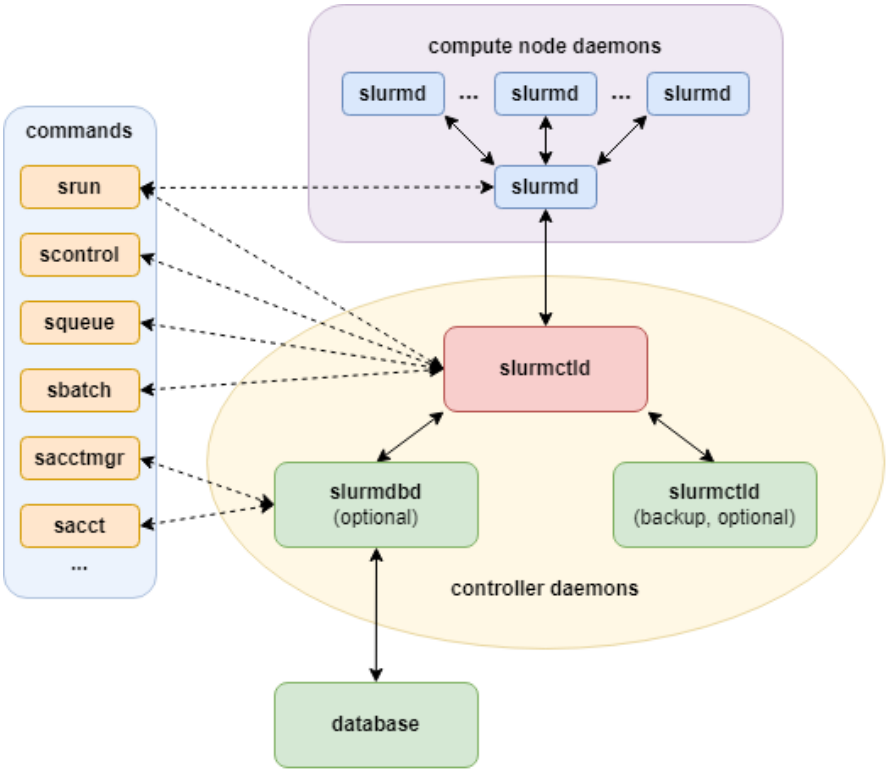


Figure 1.1 SLURM components [SchedMD, 2021].

slurmctld (the controller) is tasked with monitoring each node through communication with **slurmd**. Nodes are assessed (their configuration and availability) and grouped into partitions based on a predefined configuration file. Once a user request is made, **slurmctld** based on the current queue and job priority acts as a job manager and assigns the task to specific nodes via appropriate **slurmd**, waits for completion and subsequently launches cleanup [SchedMD, 2021].

slurmd (remote shell) runs as root on each computational node and reports to the controller its status. Depending on instructions, it can execute, manage or cleanup a job. Additionally, it handles outputs and errors and can propagate received signals

[SchedMD, 2021].

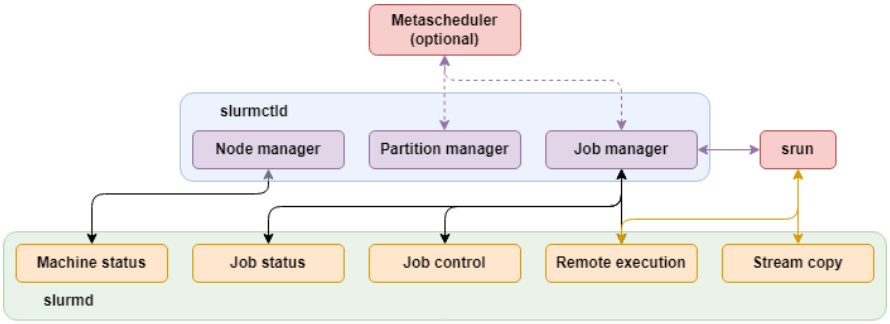


Figure 1.2 SLURM subsystems [SchedMD, 2021].

slurmdbd is an optional database daemon that can be used to collect and assign specific configuration data i.e. limits, fair-share rules, etc.

SLURM employs several commands which can be used for both management and job submission. Furthermore, job submission (execution) can be accomplished through a bash script by specifying instructions for SLURM. These instructions are differentiated from the standard code in the script through the use of keywords. Script syntax will be addressed in one of the upcoming chapters. Essential SLURM commands include:

- **scontrol** - administrator management tool
- **sacctmgr** - database management tool
- **squeue** - report job status
- **sinfo** - report system status
- **sacct** - report accounting information for a job
- **srun** - create allocation (optional) and launch a job step
- **sbatch** - submit a script for later execution (batch mode)
- **salloc** - create allocation and start a shell (interactive mode)
- **scancel** - terminate a job.

More information on commands, syntax and use will be given in the subsequent chapter.

1.3 PBS

University of Rijeka

Portable Batch System or PBS at its core performs job scheduling. Since its creation, however, it has been heavily modified and can now be extended with different plugins thus providing a comprehensive workload management system. PBS has been forked into several variants:

- **Altair PBS Professional** - commercial variant maintained by Altair
- **OpenPBS** - open-source variant
- **TORQUE** - non open-source fork of OpenPBS.

OpenPBS [OpenPBS, 2020] variant integrated into the OpenHPC stack is considered in this section.

The central element of the PBS system is the server host which loads the **server**, **scheduler** and **communication** daemons. Each computational node (execution host) runs a **MoM** daemon which is responsible for job management on the computational node. All **MoM** daemons are managed by the server host. Noted components form the PBS complex shown in Figure 1.3.

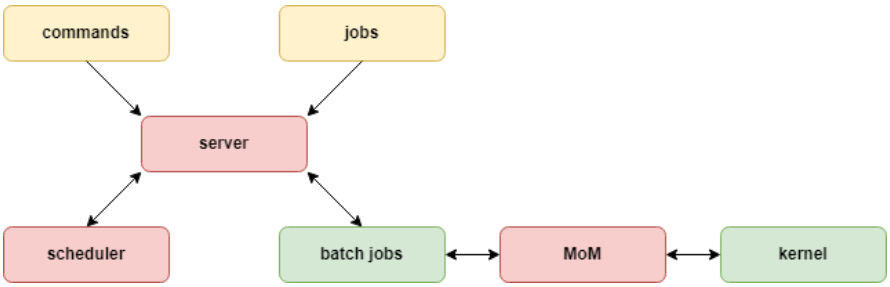


Figure 1.3 PBS complex [OpenPBS, 2020].

server acts as the central communication hub. It handles PBS commands and submits / queues jobs on the execution hosts (computational nodes).

scheduler is tasked with validating job resource requests. Job priority is assigned based on defined rules.

MoM (Machine-oriented Mini-server) manages jobs assigned to a specific computational node. This typically includes session duplication, file staging, execution and output return / cleanup. Additionally, it enables job monitoring.

Frequently used PBS commands include:

- **qsub** - submit a job
- **qdel** - terminate a job
- **qstat** - report job / queue status
- **pbsnodes** - node status.

Jobs on PBS systems are typically started via PBS scripts which contain all the necessary information for resource allocation / job execution. All lines beginning with **#PBS** are recognized as PBS directives. The syntax is similar to SLURM. More information can be found at OpenPBS [2020].

1.4 Alternative solutions

University of Rijeka

1.4.1 LSF

IBM Platform LSF (Load Sharing Facility) is a workload management tool designed for heterogeneous distributed systems, including HPC's [IBM, 2022]. It is extremely versatile and can be used on clients and compute nodes / machines that do not use Unix-like operating systems. Additionally, LSF provides a simple and comprehensive UI.

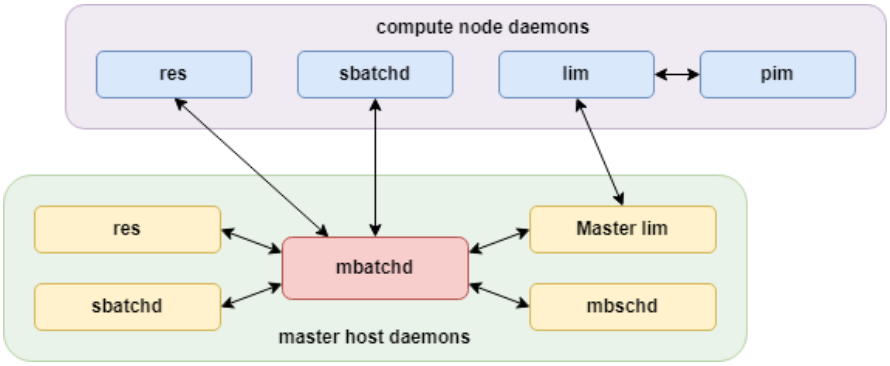


Figure 1.4 IBM Platform LSF components.

LSF relies on daemons to manage jobs. A specific node in the system is designed as the master host. The master host runs several daemons. Master batch daemon **mbatchd** is the central daemon tasked with receiving queries / commands and managing jobs. **mbschd** is the scheduler daemon. It assigns resources / priority based on defined policies and passes this information to **mbatchd**. **sbatchd** is tasked with managing jobs based on requests from **mbatchd**. **sbatchd** daemon runs on all nodes. Remote execution server **res** receives execution requests. **Master lim** and **lim** daemon processes run on the master host and compute nodes and are tasked with the collection and delivery of usage / load / configuration information for each node, which is necessary for the scheduler to adequately distribute jobs. Additionally, **lim** starts **pim** daemons which monitor each running job.

1.4.2 MOAB / TORQUE

TORQUE (Terascale Open-source Resource and Queue Manager) is a fork of the OpenPBS managed by the Adaptive Computing [Adaptive Computing, 2021b]. The underlying architecture is similar to the one presented in Figure 1.3, with additional improvements in specific areas. It features an inbuilt scheduler hence it can be classified as a standalone workload manager, however, it is typically employed as a resource manager which is paired with another scheduler. TORQUE is a component of the MOAB workload manager [Adaptive Computing, 2021a].

2. Using the SLURM workload manager

2.1 Introduction

University of Rijeka

Workload managers, specifically SLURM, will be assessed on an operational HPC system. Commands and syntax as well as scripts and examples will be presented and explained in this chapter.

All segments of an HPC will be explored in order to better familiarize users with a heterogenous HPC system. Examples and test cases introduced in this and the following chapters have been tested / executed on HPC Bura.

HPC Bura can be accessed through two secured login nodes built on the Xeon E5 architecture. Login nodes are also used to compile and install the software. The supercomputer is powered by Red Hat Enterprise Linux 7.6 and Slurm Workload Manager 17.02.11.

HPC Bura [CNRM, 2021] is a heterogeneous supercomputer owned by the University of Rijeka. Architecturally, it can be divided into three distinct parts:

- **Cluster**
- **SMP**
- **GPGPU**

Cluster section is a multicomputer system comprised of 288 compute nodes with two Xeon E5 processors per node (24 physical cores per node). A total of 6912 physical cores are available to users. Each node has 64 GB of memory and 320 GB of disk space, respectively.

SMP section is a multiprocessor system with a large amount of shared memory. SMP is made up of 16 Xeon E7 processors with a total of 256 physical cores, 6 TB of memory and 245 TB of local storage. Two nodes are available.

GPGPU section is comprised of four nodes with two Xeon E5 processors (16 physical cores per node) and two NVIDIA Tesla K40 general-purpose GPUs per node.

Important Note

In order to connect to HPC Bura, a VPN connection must be established. Please contact your HPC provider in order to determine if there are any specific / additional requirements to connect to their HPC.

2.2 Commands

University of Rijeka

SLURM offers many commands for job submission / resource allocation and job management. The availability of some commands, however, can vary depending on SLURM implementation specificities. For example, accounting commands and access to the relevant data can either be restricted or the accounting can simply be omitted / not monitored (**slurmdbd** daemon inactive). In view of that, in this section, we will focus on commands which are essential and should be available in every HPC system.

sinfo

Command **sinfo** is typically used to collect information on available partitions and respective nodes. Partition is a group of compute nodes and is defined based on e.g. configuration / architectural equivalence or any other arbitrary reason / requirement. Nodes grouped into a partition share limits and policies. Creating partitions defines separate queues to which jobs can be submitted. This can be useful for separating e.g. commercial and non-commercial users. Partitions can also overlap.

```
[user@bura2 ~]$ sinfo
PARTITION      AVAIL  TIMELIMIT  NODES  STATE NODELIST
computes_thin*    up   infinite     10 drain* bura[110-119]
computes_thin*    up   infinite     10  down* bura[100-109]
computes_thin*    up   infinite     10  drain bura[120-129]
computes_thin*    up   infinite      5    mix bura[130-134]
computes_thin*    up   infinite      5  alloc bura[135-139]
computes_thin*    up   infinite    248   idle bura[140-387]
guest             up 7-00:00:00     10 drain* bura[110-119]
guest             up 7-00:00:00     10  down* bura[100-109]
guest             up 7-00:00:00     10  drain bura[120-129]
guest             up 7-00:00:00      5    mix bura[130-134]
guest             up 7-00:00:00      5  alloc bura[135-139]
guest             up 7-00:00:00    248   idle bura[140-387]
comp_gpu          up   infinite      4   idle bura[500-503]
comp_smp          up   infinite      2   idle bura[36,44]
```

sinfo command can be modified with different options. For example, **-s** allows for a succinct overview of partitions and node statuses.

```
[user@bura2 ~]$ sinfo -s
PARTITION      AVAIL  TIMELIMIT   NODES(A/I/O/T)  NODELIST
computes_thin*    up   infinite    10/248/30/288  bura[100-387]
guest             up 7-00:00:00    10/248/30/288  bura[100-387]
comp_gpu          up   infinite          0/4/0/4  bura[500-503]
comp_smp          up   infinite          0/2/0/2  bura[36,44]
```

In order to view nodes based on their status, option **-t** followed by the status of interest can be useful. Additionally, we can track e.g. idle nodes in a specific partition by invoking option **-p** and specifying a partition.

```
[user@bura2 ~]$ sinfo -t IDLE -p comp_SCjobs
PARTITION   AVAIL  TIMELIMIT  NODES  STATE NODELIST
comp_SCjobs    up   infinite      8   idle bura[180-187]
```

squeue

squeue displays information on all jobs currently in queues, regardless of whether the jobs are running, pending, or waiting for a specific trigger (e.g. dependency).

```
[user@bura2 ~]$ squeue
JOBID PARTITION NAME  USER ST  TIME NODES NODELIST(REASON)
22807  comp_smp job7  user PD  0:00     2 (Resources)
22801 computes_ job1  user  R 48:59     1 bura160
22802  comp_smp job2 userA  R 35:11     1 bura36
22803  comp_gpu job3 userB  R 13:09     1 bura500
22804 computes_ job4 userA  R 11:05     1 bura161
22805 computes_ job5  user  R  8:52     4 bura[162-165]
22806 computes_ job6  user  R  5:52     1 bura166
```

Typically, **squeue** command is used with option **-u** followed by a username. This will display all jobs for a specific user.

```
[user@bura2 ~]$ squeue -u userA
JOBID PARTITION NAME  USER ST  TIME NODES NODELIST(REASON)
22802  comp_smp job2 userA  R 35:11     1 bura36
22804 computes_ job4 userA  R 11:05     1 bura161
```

Similarly, option **-p** followed by the partition name and **-t** followed by a job state code (e.g. R, PD) will display all jobs running on a specific partition and all jobs with a specific state, respectively.

sdiag

Command **sdiag** is usually not interesting to users as it provides detailed information on the scheduler, including statistics. This includes e.g. job counts and **slurmctld** thread and agent counts.

```
[user@bura2 ~]$ sdiag
*******************************************************
sdiag output at Tue Jan 1 12:21:44 2022
Data since      Tue Jan 1 01:00:00 2022
*******************************************************
Server thread count: 3
Agent queue size:    0

Jobs submitted: 8665
Jobs started:   8666
Jobs completed: 8563
Jobs canceled:  129
Jobs failed:    0

Main schedule statistics (microseconds):
Last cycle:   164
Max cycle:    31986
Total cycles: 11362
Mean cycle:   628
Mean depth cycle:  0
Cycles per minute: 15
Last queue length: 0
...
```

scontrol

scontrol can be used to display status data and information on specific SLURM / HPC components. System administrators can use **scontrol** to modify SLURM configuration (requires proper permissions). For example, command **scontrol show partition $partition_name** will show information on a specific partition.

```
[user@bura2 ~]$ scontrol show partition comp_smp
PartitionName=comp_smp
AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
AllocNodes=ALL Default=NO QoS=N/A
DefaultTime=NONE DisableRootJobs=NO ExclusiveUser=NO GraceTime=0 Hidden=NO
MaxNodes=UNLIMITED MaxTime=UNLIMITED MinNodes=1 LLN=NO MaxCPUsPerNode=UNLIMITED
Nodes=,bura36,bura44
PriorityJobFactor=1 PriorityTier=1 RootOnly=NO ReqResv=NO OverSubscribe=NO
OverTimeLimit=NONE PreemptMode=OFF
State=UP TotalCPUs=512 TotalNodes=2 SelectTypeParameters=NONE
DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED
```

Furthermore, **scontrol show nodes $node_name** can be used to gather information on a specific node. Command **scontrol** is versatile. It can be used to display the specifics of a job.

```
[user@bura2 ~]$ scontrol show job 22801
JobId=22801 JobName=job1
UserId=user(1010) GroupId=user(1010) MCS_label=N/A
Priority=4294542294 Nice=0 Account=(null) QOS=normal
JobState=RUNNING Reason=None Dependency=(null)
Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
RunTime=00:46:64 TimeLimit=00:120:00 TimeMin=N/A
SubmitTime=2022-01-01T11:31:23 EligibleTime=2022-01-01T13:31:23
StartTime=2022-01-01T11:31:23 EndTime=2022-01-01T13:31:23 Deadline=N/A
PreemptTime=None SuspendTime=None SecsPreSuspend=0
Partition=computes_thin AllocNode:Sid=bura160:31216
ReqNodeList=(null) ExcNodeList=(null)
NodeList=bura160
BatchHost=bura160
NumNodes=1 NumCPUs=22 NumTasks=22 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
TRES=cpu=22,mem=22G,node=1
Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
MinCPUsNode=1 MinMemoryCPU=1G MinTmpDiskNode=0
Features=(null) DelayBoot=00:00:00
Gres=(null) Reservation=(null)
OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
Command=/home/user/run.sh
WorkDir=/home/user
StdErr=/home/user/log.out
StdIn=/dev/null
StdOut=/home/user/log.out
Power=
```

By including option **-dd**, the batch script for a specified job will additionally be displayed. Finally, **scontrol** can be used to manage a job.

```
[user@bura2 ~]$ scontrol hold 22801
[user@bura2 ~]$ scontrol release 22801
[user@bura2 ~]$ scontrol suspend 22805
[user@bura2 ~]$ scontrol resume 22805
[user@bura2 ~]$ scontrol requeue 22805
```

srun

srun is used to start one or more tasks inside an allocation (resources are inherited). It is typically used in conjunction with **salloc** or inside a **sbatch** script. If no resources are allocated, **srun** will create an allocation (minimal) or one can be specified by the user.

```
[user@bura2 ~]$ srun -N 2 -n 48 -t 30:00 ./myjob
```

salloc

Command **salloc** is used to request allocation for an interactive job. Once requested resources are approved, it is necessary to connect to the approved allocation.

```
[user@bura2 ~]$ salloc --nodes=1 --time=60:00 --partition=computes_thin
salloc: Granted job allocation 22851
[user@bura2 ~]$ srun --jobid=22851 --pty /bin/bash
[user@bura155 ~]$ hostname
bura155
[user@bura155 ~]$ exit
exit
[user@bura2 ~]$ exit
exit
salloc: Relinquishing job allocation 22851
```

If connected to an allocation (node), it is assumed that input will occur, otherwise, connection will be terminated (connection only, allocated resources will be reserved until time expires). A more straightforward allocation can be done in a single line.

```
[user@bura2 ~]$ salloc -N 1 -t 60:00 -p computes_thin srun --pty bash
salloc: Granted job allocation 22855
[user@bura156 ~]$ exit
exit
salloc: Relinquishing job allocation 22855
```

Finally, **srun** command can be used to execute a job without the need to connect to the allocation. If **srun** is not used, code will be executed on the current node (usually login node).

```
[user@bura2 ~]$ salloc -N 3 -t 60:00 -p computes_thin
salloc: Granted job allocation 22860
[user@bura2 ~]$ srun /bin/hostname
bura155
bura156
bura157
[user@bura2 ~]$ /bin/hostname
bura2
```

sbatch

sbatch submits a batch script to SLURM. Typically, all relevant resource requests and jobs are defined in the script. SLURM scripts will be discussed in section 2.3.

```
[user@bura2 ~]$ sbatch slurm_script.sh
```

scancel

Command **scancel** cancels a job or a job step (this includes allocations).

```
[user@bura2 ~]$ scancel 22860
```

Similarly to all other commands, **scancel** can have multiple options, thus it is possible to cancel e.g. multiple jobs on a given partition started by the user.

```
[user@bura2 ~]$ scancel -u user -p computes_thin
```

sreport

sreport generates reports from job accounting data (including utilization statistics). Accounting must be enabled and available (active **slurmdbd** daemon). The command can be used to report usage for a specific user in a set period.

```
[user@bura2 ~]$ sreport user topusage start=1/1/22 end=1/10/22 -t hours users=user
```

sacct

sacct can be used to retrieve and display accounting data for all jobs in the database. Accounting must be enabled and available (active **slurmdbd** daemon). The command can be used to e.g. inspect job history.

```
[user@bura2 ~]$ sacct -X -u user
```

sacctmgr

Command **sacctmgr** can be used to display and modify SLURM account information. Accounting must be enabled and available (active **slurmdbd** daemon).

```
[user@bura2 ~]$ sacctmgr -s show user name=user
```

2.3 Scripts

University of Rijeka

SLURM jobs are typically submitted through a batch script which is often referred to as a submission script. Submission scripts are simple shell scripts and include specific commands / parameters that are to be interpreted by SLURM. These commands describe resource requests and specify additional options / requirements. Following the resource allocation, specific code / commands / program call noted in the script is executed. A script example is given below.

```
#!/bin/bash
#SBATCH --job-name=job
#SBATCH --output=output.out
#SBATCH --time=00:10:00
#SBATCH --partition=computes_thin
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1G

srun hostname
```

Slurm directives (lines beginning with **#SBATCH**) should follow the **#!/bin/bash**. Script is executed by using the previously described **sbatch** command.

```
[user@bura2 ~]$ sbatch script.sh
```

Important Note

Submission script executed with **sh script.sh** will run on the login node. This is an improper way of submitting a job.

2.3.1 Resource requests

Memory and processors are typically the most important resources in an HPC system. The allocation of these resources is governed by different parameters.

The parameter **--ntasks**, **-n** is used to define the number of tasks (processes) for a distributed job. By default, each process is assigned a single processor. SLURM will manage the distribution of tasks unless directly specified with **--ntasks-per-node**. Additionally, requested tasks can be distributed across nodes by specifying the number of nodes with **--nodes**, **-N**. This approach, however, does not guarantee equal distribution of tasks. **--ntasks-per-gpu** is relevant for GPU nodes and specifies the number of tasks to be invoked per each GPU.

```
#SBATCH --nodes=3
#SBATCH --ntasks=30
```

Noted script excerpt will request resources for 30 tasks. The distribution of these tasks is governed by SLURM. Distribution can also be conducted manually.

```
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=10
```

The parameter **--cpus-per-task**, **-c** facilitates the definition of multiple processors for each task. SLURM will ensure that processors for each task are located on the same node. This approach is common for OpenMP problems.

```
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=28
```

Memory requests can be defined with one of three parameters: **--mem**, **--mem-per-cpu** and **--mem-per-gpu**. **--mem** defines required memory per node. Other parameters are self-explanatory. Specifying memory requests is encouraged as otherwise, depending on the HPC system, this might imply that the entire node is necessary i.e. should be allocated. If **--mem** is set to zero, the entire node will be allocated.

```
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=4
#SBATCH --mem-per-cpu=1024MB
```

Apart from processors, users might need to access GPUs. The required number of GPUs can be specified with **--gpus**, **-G**. Additionally, necessary GPUs can be defined on a per-node (**--gpus-per-node**) or per-task basis (**--gpus-per-task**). More commonly, however, GPUs are requested as generic resources using the **--gres**. The parameter is followed by an argument that defines the resource and necessary amount e.g. **--gres=gpu:2**.

2.3.2 Common parameters

Job name can be defined with **--job-name**, **-J**. Defining a name enables easier identification / tracking of a job while in a queue.

```
#SBATCH --job-name=job
```

Standard output i.e. output and errors of a job can be redirected to a specific file with **--output**, **-o** parameter.

```
#SBATCH --output=output.out
```

Specifying a job time limit is important as it prevents unresponsive or bad-behaving jobs from unnecessarily keeping resources allocated for a longer period. If not defined, it defaults to the partition time limit. Parameter **--time**, **-t** accepts most standard time formats.

```
#SBATCH --time=00:10:00
```

Jobs can request resources from specific segments of an HPC system called partitions. The parameter **--partition**, **-p** indicates which resources is SLURM to allocate (which resource pool to use). If not defined, the default partition will be utilised.

```
#SBATCH --partition=computes_thin
```

Jobs can be executed exclusively on requested resources (nodes) with **--exclusive**. This means that other jobs or users can not share/utilise node(s) which are allocated to the current job.

```
#SBATCH --exclusive
```

SLURM allows explicit specification of hosts (compute nodes) to be used for a specific job. The parameter **--nodelist**, **-w** followed by the hostnames indicates which nodes to use. If needed, additional nodes will be utilised so as to satisfy the defined resource request.

```
#SBATCH --nodelist=bura[100-110]
```

Arrays (**--array**, **-a**) can be used to quickly execute multiple similar job instances using the same job parameters. If it is necessary e.g. to parse 1000 textual files named **file_$NUMBER.txt** where **NUMBER** is an integer from 1 to 1000, arrays can be used to directly submit jobs which will parse said files. It is necessary, however, to specify (use) proper array variables. In noted example, **NUMBER** should be replaced with **SLURM_ARRAY_TASK_ID**.

```
        ...
#SBATCH --array=0-9
        ...
echo Task $SLURM_ARRAY_TASK_ID
python fileditor.py $SLURM_ARRAY_TASK_ID
```

2.4 Examples

University of Rijeka

2.4.1 Shared memory examples

Shared memory problems are executed on a single node and spawn a single task that can utilise multiple processors. OpenMP jobs are a prime example of shared memory parallelism within a single node. This parallelism concept is known as multithreading.

Resource-demanding OpenMP jobs on HPC Bura are typically executed on the SMP partition (**comp_smp**). Simpler jobs can be executed on cluster nodes (**computes_thin**). Environment variable **OMP_NUM_THREADS** should be set and should equal the number of processors requested for the task (**SLURM_CPUS_PER_TASK**).

```
#!/bin/bash
#SBATCH --job-name=omp
#SBATCH --output=omp.out
#SBATCH --time=00:60:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=12
#SBATCH --mem-per-cpu=1024M
#SBATCH --partition=computes_thin

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./omp_problem
```

```
#!/bin/bash
#SBATCH --job-name=smp_omp
#SBATCH --output=smp_omp.out
#SBATCH --time=24:00:00
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=128
#SBATCH --mem=4T
#SBATCH --partition=comp_smp

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
./complex_omp_problem
```

2.4.2 Distributed memory examples

Programs that can utilise distributed memory typically rely on Message Passing Interface (MPI) to achieve communication between distributed resources. MPI jobs are executed on the cluster (**computes_thin**).

```
#!/bin/bash
#SBATCH --job-name=mpi
#SBATCH --output=mpi.out
#SBATCH --time=12:00:00
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=24
#SBATCH --mem-per-cpu=1024M
#SBATCH --partition=computes_thin

srun ./mpi_problem
```

Usually, however, it is necessary to prepare the environment for the software that is to be used. HPC systems utilise environment modules to properly configure the user's shell environment. This approach enables compartmentalization and coexistence of different program variants (e.g. different MPI versions / implementations can be installed and can be loaded on request).

Environment modules on HPC Bura are deployed on login nodes and are not available on compute nodes hence users must prepare the environment prior to submitting a job (if available on compute nodes, modules can be loaded inside the submission script). All available modules can be inspected with **module avail**.

```
module load OpenFOAM_8/OF8_IntelMPI-2019.8_gcc10.2_OPT
```

The noted module will load OpenFOAM 8 as well as GCC 10.2 and Intel MPI 2019.8. MPI job can now be submitted using either **srun** or **mpirun**.

```
#!/bin/bash
#SBATCH --job-name=mpi
#SBATCH --output=mpi.out
#SBATCH --time=12:00:00
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=24
#SBATCH --mem-per-cpu=1024M
#SBATCH --partition=computes_thin
#SBATCH --exclusive

sed -i "s/numberOfSubdomains.*[0-9][0-9]*;/numberOfSubdomains $SLURM_NTASKS;/g" \
system/decomposeParDict

decomposePar > log.decomposePar

mpirun -np $SLURM_NTASKS pimpleFoam -parallel > log.LOGGIT 2>&1
```

2.4.3 GPU jobs

A commercial Lattice Boltzmann CFD code ultraFluidX utilises GPU accelerators. The environment can be configured by loading the appropriate environment module.

```
module load ultraFluidX
```

ultraFluidX can now be executed with **mpirun** where the number of processors must be larger than the number of GPUs (**--gpus** + 1).

```
#!/bin/bash
#SBATCH --job-name=gpu
#SBATCH --output=gpu.out
#SBATCH --time=12:00:00
#SBATCH --nodes=1
#SBATCH --ntasks=2
#SBATCH --mem-per-cpu=1024M
#SBATCH --partition=comp_gpu

mpirun -np 2 ultraFluidX problem.xml -o output
```

2.4.4 Hybrid problems

Hybrid MPI / OpenMP jobs rely on OpenMP for intra-nodal communication (inside each SMP node) while MPI facilitates inter-nodal communication. MPI environment module must be loaded prior to running the script.

```
module load mpi/openmpi-4.1.1
mpicc -fopenmp hybrid_problem.c -o hybrid_problem
```

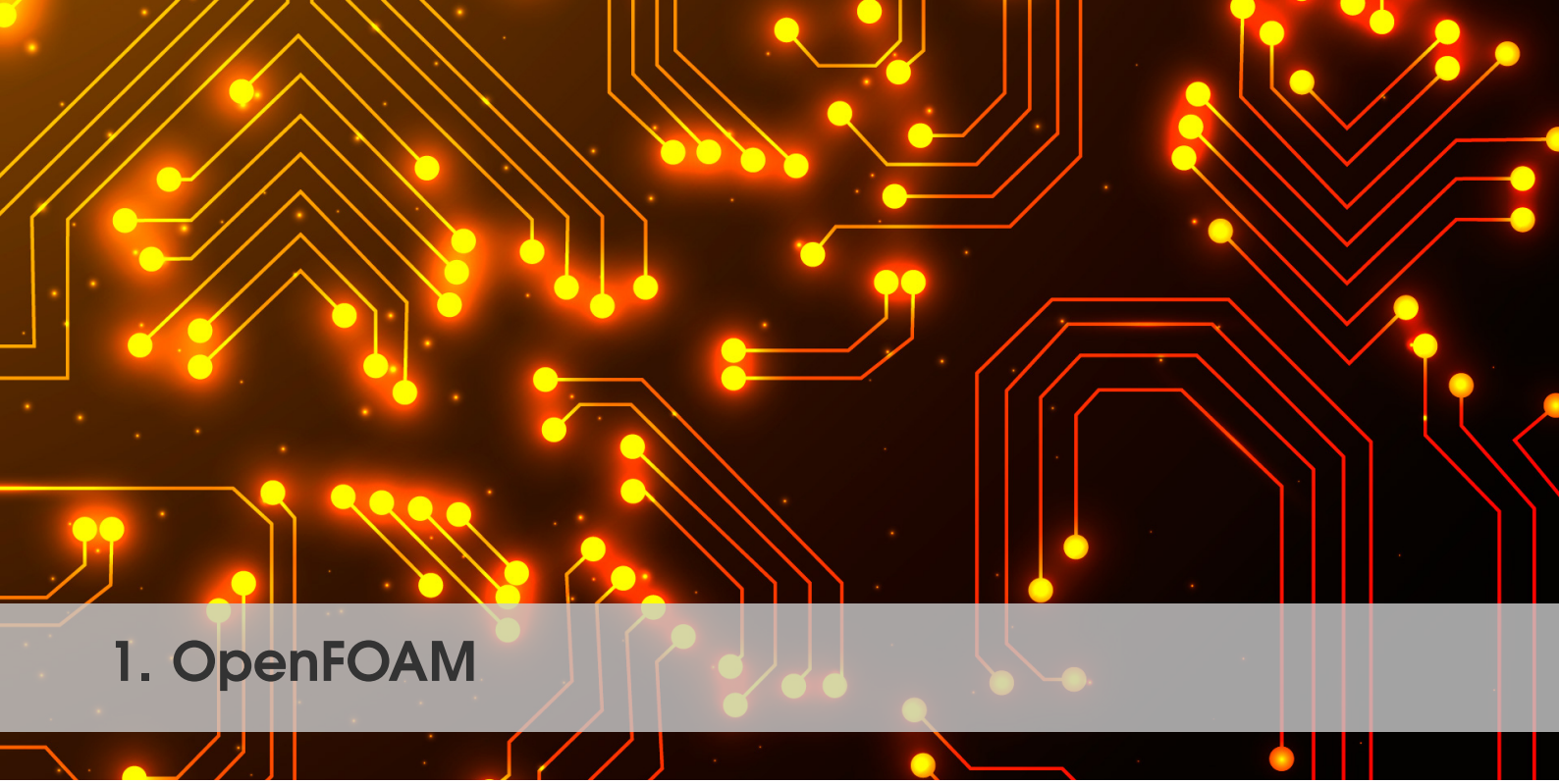
If executing a custom code, code must be compiled with proper options (e.g. **-fopenmp**) due to included OpenMP directives.

```
#!/bin/bash
#SBATCH --job-name=hybrid
#SBATCH --output=hybrid.out
#SBATCH --time=12:00:00
#SBATCH --ntasks=6
#SBATCH --cpus-per-task=10
#SBATCH --mem-per-cpu=1024M
#SBATCH --partition=computes_thin

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
mpirun ./hybrid_problem
```

Part III

Problems and examples



1. OpenFOAM

1.1 Introduction

Technical University of Denmark

OpenFOAM includes a versatile range of solvers that can be used for common transport problems (e.g. fluid flow inside simple domains) or, for highly specific use cases, can be implemented by defining the mathematical model (e.g. simulation of large-scale fermentations with tracking of individual cells). In other words, OpenFOAM can be adapted to individual needs. Backed by an active community, it is also an attractive choice for CFD beginners.

OpenFOAM is deployed as a Linux package. Although installation and use are possible under other operating systems, it is recommended to have OpenFOAM installed on Linux. This has several advantages. First, Linux will allow a more streamlined CFD workflow. Several core routines are native to the Linux environment and are conveniently accessed from the Linux command line. Repetitive tasks can be effectively reduced through shell scripting, which also contributes to building better programming habits. Second, the more complex a CFD simulation gets, the more crucial it will become to simulate on a high-performance computing (HPC) cluster. In this sense, by developing on Linux, the transition to the HPC ecosystem should be rather simple.

It is worth mentioning that Linux evolved from Unix, the early operating system that was originally *designed* for HPC applications. Therefore, Linux-based operating systems are commonplace when it comes to managing HPC clusters. To illustrate; the world's top 500 fastest supercomputers all run on Linux and there are no signs that this will change anytime soon. If you are interested in running CFD calculations on HPC at some point, it is sensible to incorporate Linux in your workflow right from the start. As a rule of thumb, a personal computer might be sufficient to solve simple CFD problems for mesh sizes of up to 10^5 cells per core. For use cases involving e.g. 10^6 cells or more, you will want to migrate to an HPC cluster to perform the CPU-intensive calculations.

There are several options to set up OpenFOAM on your personal computer. For example, you might install Linux as your primary operating system and install OpenFOAM on top of that. Admittedly, this would require a rather drastic change to your system, especially if you are only starting. So unless you have a spare computer lying around, this is typically not feasible for most novel users. Another option is the use of pre-compiled container packages (e.g. Docker image). OpenFOAM provides container images for many operating systems, including Windows, MacOSX, and Linux. A major advantage of this approach is the simple installation procedure. However, there are also disadvantages. Container images require another software layer for program execution and this is adding to the system load. The container packages also lack visualization tools, such as ParaView. Due to performance limitations, it is recommended to post-process simulation data with a native installation of ParaView rather than doing so directly in the container image. Unfortunately, this is disrupting the CFD workflow and provides only a little space to acquire some Linux basics. A third installation option - and the preferred one for this tutorial - is the use of a Virtual Machine (VM). In this way, an entire Linux environment can be created within your running host system, be it Windows, MacOSX, or different. It will provide the user with a safe test bed for the OpenFOAM installation and the simulation cases.

In the first part of this tutorial, we will focus on building a minimal working CFD

setup on your local computer. To this end, you will create a Linux environment inside a VM and install OpenFOAM under Linux.

1.2 Creating a Linux environment on your computer

We will now focus on building a Linux environment on your personal computer. Should you already have access to a Linux environment, you can reasonably skip this part. However, considering that this tutorial is targeted particularly at first-time users, detailed instructions are provided on how to set up such an environment on a Windows or MacOSX system. We will use *VirtualBox* to set up a Linux environment inside a VM. The chosen Linux distribution is *Debian with XFCE desktop environment*.

Important Note

Make sure that you have at least 8 GB of RAM installed on your computer and at least 25 GB of free disk space. You should have internet access. Ensure that you have hardware virtualization VT-x/VT-d or AMD-v enabled in your BIOS settings.

1.2.1 Download and install VirtualBox

VirtualBox is an open-source hypervisor used to virtualize operating systems within an existing host. The virtualized system is known as Virtual Machine (VM). VMs provide a safe sandbox for software developments without affecting the host system. The CFD simulation described in this tutorial will be developed inside a Debian (Linux) VM. Therefore, as a first step, download the latest version of VirtualBox at https://www.virtualbox.org/wiki/Downloads. At the time of writing, VirtualBox 7.0.0. is the latest version. Select the installer version that matches your host system, e.g. Windows. Once downloaded, install VirtualBox. Detailed information on the installation procedure is available at https://www.virtualbox.org/manual/.

1.2.2 Download a Debian image

Debian is a stable, open-source desktop operating system based on the Linux kernel. Download the latest iso-image for Debian with the Xfce desktop environment at . At the time of writing, the latest stable Debian release is Debian 11.5. If you plan on deploying it on a 64-bit PC (amd64), the selected iso-file may read as *debian-live-11.5.0-amd64-xfce.iso*.

Once the download is complete, verify the file's checksum. Even though it is generally a good practice to do so, please note that this step is not strictly required. However, it will allow you to confirm the integrity of your download and may prevent you from dealing with corrupt or broken files. On a Windows system, you can use the built-in **certUtil** command. Proceed as follows:

- Open the directory that contains the iso-image.
- Click on the address bar and type **cmd**. This will bring up a command prompt.
- In the command prompt, run **certUtil -hashfile <FILE_NAME> SHA256**, where **<FILE_NAME>** has to be replaced with the exact name of the iso file you downloaded.

The command will return the file's SHA256 checksum. Compare this to the official SHA256SUMS file which is provided alongside the iso-images at . The calculated checksum and the official checksum must match.

1.2.3 Create a new Virtual Machine

Start VirtualBox and click the *New* button. A dialog box appears (see Figure 1.1). In the **Name and Operating System** section, type *Debian* in the **Name** field, set **Type** to *Linux* and **Version** to *Debian (64-bit)*. As for the ISO Image, select the Debian iso file which you downloaded in the previous step. Make sure to check the **Skip Unattended Installation** option.

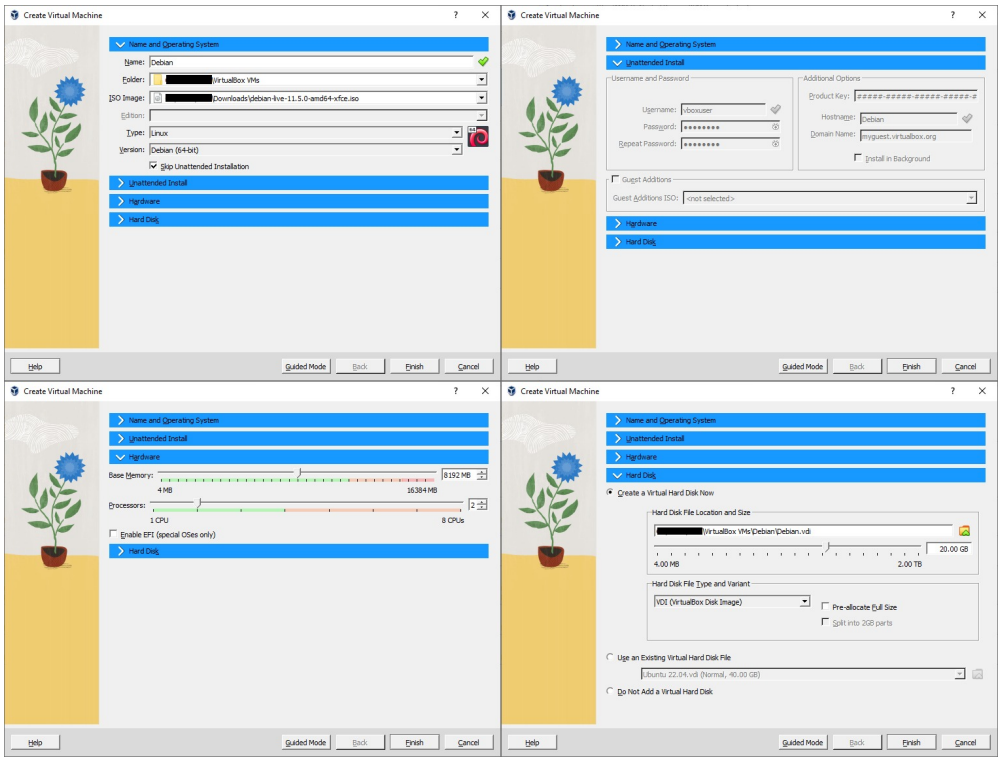


Figure 1.1 Configuring a Virtual Machine.

Head over to the **Hardware** section. Set the **Base Memory** to at least *4 GB (4096 MB)*. For better performance of the VM, increase the memory size as long as the sliding indicator remains in the green range. Please keep in mind that the amount of memory selected here will be taken away from the host machine and presented to the VM. Therefore, setting this number too high will decrease the host system's performance as long as the VM is running. Select the number of **Processors**. The larger the number of processors, the more CPU resources will be allocated to the VM. Again, move up the slider, but make sure that the slider remains in the green range.

In the **Hard Disk** section, create a Virtual Hard Disk of the *VDI (VirtualBox Disk Image)* type and set **Hard Disk Size** to least 25 GB. Click on *Finish*. You will now return to the VirtualBox main window.

1.2.4 Installing Debian on the VM

In the main window, select the Debian VM and click the *Start* button. The VM is now powering up. From Debian's boot menu, select *Graphical Debian Installer* using the

arrow keys and hit *Enter*. Follow the installation procedure. Specify your language, location and keyboard settings. Next, set **Hostname** to *debian* and **Domain Name** to *localhost* – for a personal VM. Choose a **Root Password** and remember it. Define a **User Name** and a **User Password**. For the sake of this tutorial, *cfd* was chosen as the <username> and *debian* as the <password>. Remember your credentials to avoid losing access to the VM at a later time. As you proceed, select the correct time based on your location.

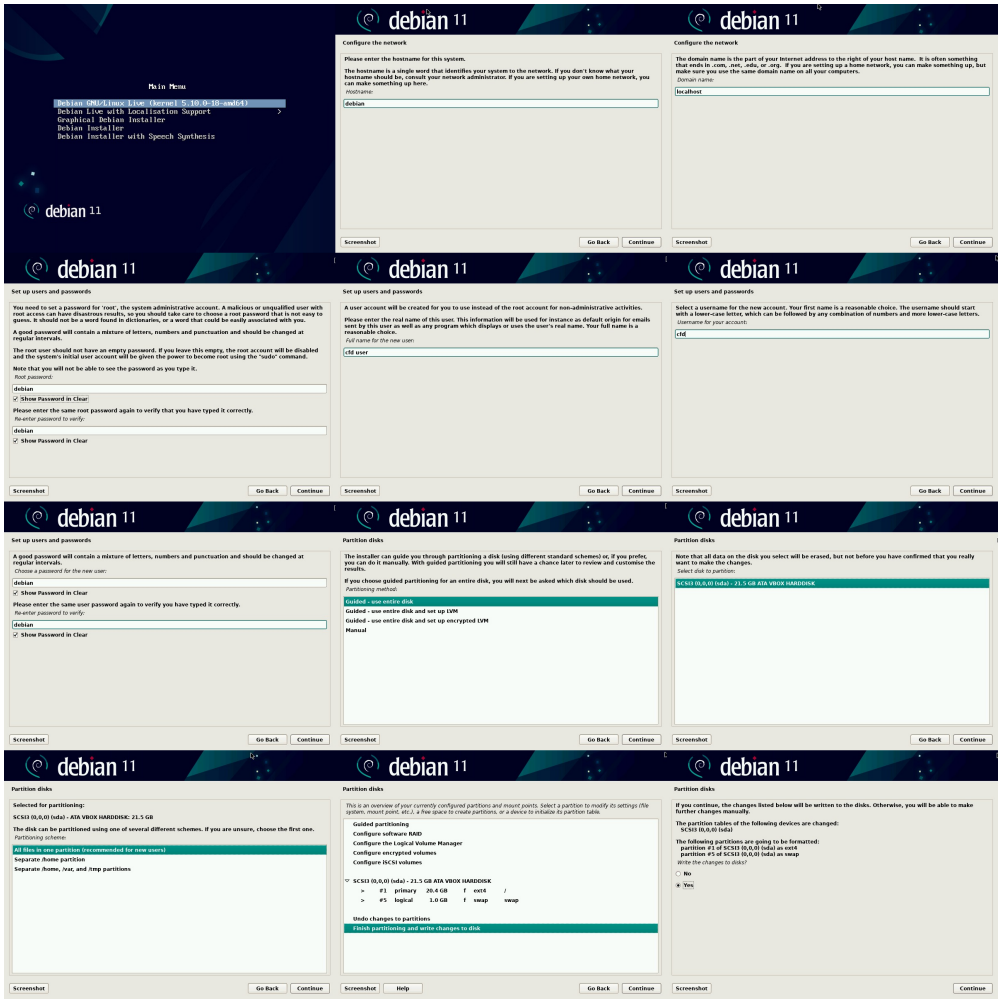


Figure 1.2 Snapshots of the Debian installation process. Pictures are arranged on a grid, following the actual installation sequence. For each row, pictures are ordered from left to right, and rows are ordered from top to bottom.

In the next step, you will partition the hard drive for the Debian installation. This will not affect your host system. In the **Partition disks** dialog, choose *Guided – use entire disk* and click *Continue*. You should now select the virtual hard disk that was created during the VM setup (see Figure 1.2), i.e. the *VBOX* hard disk with a size of 25 GB. Click *Continue*. For the partitioning scheme, select *All files in one partition* and click *Continue*. Select *Finish partitioning and write changes to disk* and click *Continue*. Tick *Yes* when asked to write changes to disk. Click *Continue*. The installation will start. The process takes a couple of minutes.

Once this installation has stopped, you will have to **Configure the package manager**. When asked to use a network mirror tick *Yes* and click *Continue*. Select *deb.debian.org* as

the **Debian archive mirror** and click *Continue*. Leave the **HTTP proxy information** blank. Click *Continue*. When asked to **Install the GRUB boot loader to your primary drive** tick *Yes* and press *Continue*. Now, select the **Device for boot loader installation**. There is only one hard drive available in the selection menu (i.e. */dev/sda*). Select this drive and press *Continue*. The installation will now finish. Noted process is shown in Figure 1.3.

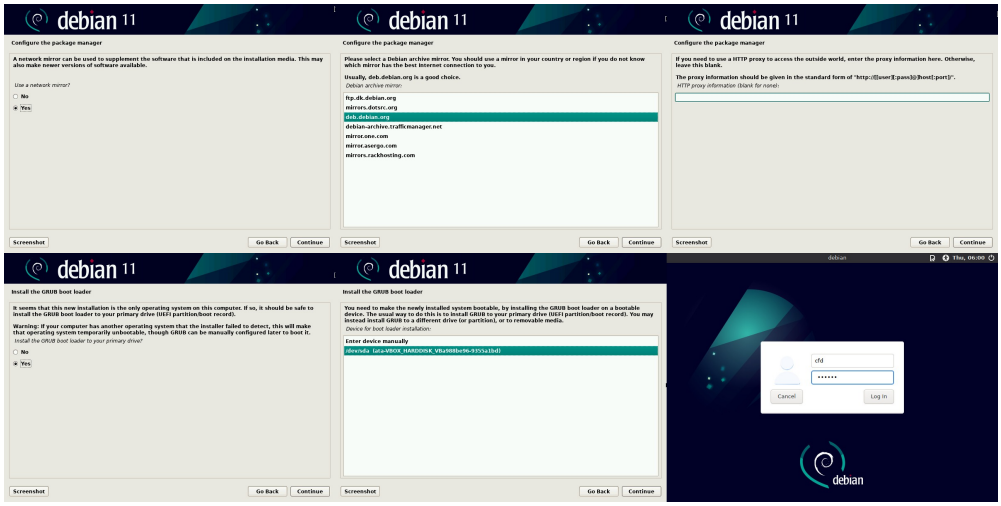


Figure 1.3 Snapshots of the Debian installation process. Final steps in the process.

1.2.5 Configuring the VM

The system will reboot automatically after installation. A login screen appears. Enter the username and password as defined during the installation process. Click on *Log In*. A desktop environment will come up, indicating that the OS is now up and running. At this time, you might experience a low screen resolution (800x600). To enable higher screen resolutions, we will now install VirtualBox Guest Additions. Make sure the VM window is in focus and press *Ctrl-Alt-T* to bring up a terminal window. Alternatively, you can click the terminal icon at the bottom. Type the following commands in the terminal window and complete each line by pressing the *Enter* key.

```
su
sudo usermod -aG sudo <username>
su <username>
```

The **su** command will prompt you for a password. Type the *root password* and confirm with the *Enter* key. This will sign you in as the system's superuser (or root). The root user is a special user account that has unrestricted read and write privileges to all areas of the operating system. The root user can also add other user accounts to the *sudo* group. Users in that group benefit from the same root privileges should the need arise. Therefore, before executing **sudo usermod -aG sudo <username>**, replace <username> with your actual username (e.g. *cfd*) so as to add your personal user account to the *sudo* group. Finally, **su <username>** will switch you back from root to your normal user account. It is now time to install some prerequisite packages:

```
sudo apt update -y && sudo apt upgrade
sudo apt install build-essential dkms linux-headers-$(uname -r)
```

The **sudo** command grants the personal user account (e.g. *cfd*) temporary root privileges. The first command will thus update the package information. Type the first command line and press *Enter*. Confirm by entering your normal *user password*. When prompted for your approval to continue, type *Y* and press *Enter*. Wait until the update is complete. The second command will install the build essentials package, DKMS framework, and kernel headers. Type the second command line in the terminal and press *Enter*. Continue as before.

To install VirtualBox Guest Additions, proceed as follows. From the menu bar in the Debian VM window, select *Devices -> Insert Guest Additions CD image*. A disc icon appears on the desktop. Install guest additions by entering the following commands in the terminal:

```
sudo mkdir -p /mnt/cdrom
sudo mount /dev/cdrom /mnt/cdrom
cd /mnt/drom
sudo ./VBoxLinuxAdditions.run
```

Once the installation is complete, type **sudo reboot** and press *Enter*. The system will now restart and changes should take effect. Log in with your user credentials. From the menu bar in the Debian VM window, select *View -> Auto-resize Guest Display*. You can now maximize the VM window and the VM screen should adapt to full resolution. It is also useful to allow copy-pasting between the host system and the VM. To do so, select *Devices -> Shared Clipboard -> Bidirectional*.

Now, bring up a terminal and run **sudo usermod -aG vboxsf <username>**. This will add your user account to the *vboxsf* user group, which is necessary to create a shared directory between the host system and the VM. This will come in handy in many situations, for example, when exporting simulation results to the host system. From the VM menu, select *Devices -> Shared Folders -> Shared Folder Settings*. In the pop-up menu, click the *Add folder (plus)* button on the right. Select a **Folder Path** that you would like to become the shared directory on your host system. Assign a **Folder Name** (e.g. *share*) and select the **Auto-mount** and **Make Permanent** option. Click OK. After the system reboot, the shared directory can be accessed as `/media/sf_share/` where the chosen directory name has been prepended by `sf\_`.

1.2.6 Installing OpenFOAM

At this point, you should have a Linux system running on your computer. It is now time to install OpenFOAM. Several Linux distributions offer pre-compiled software packages for OpenFOAM as well as for accompanying software such as ParaView. Nevertheless, depending on the specific distribution (e.g. Debian, Ubuntu, CentOS), it may be necessary to compile OpenFOAM from the source. Reasons for taking this approach might be that the available software builds in the repository are outdated and you wish to upgrade to a more recent version. Another reason might be that you desire to install a very specific version of OpenFOAM so as to create identical simulation setups on two or more computers, for example, if you wish to ensure unified simulation behavior among your local computer and a cloud server operating on the same simulation.

In this tutorial, we are going to build OpenFOAM and ParaView from source. We will use the latest source packages provided by the *OpenFOAM Foundation*. The source code is hosted at `https://github.com/openfoam` and can be downloaded, compiled and run on any Linux-based operating system. The latest stable release at the time of

writing is OpenFOAM 10. Open a terminal and execute the following commands. If prompted, confirm the installation with *Y* and *Enter*.

```
# updates the package lists
sudo apt-get update

# Install required packages for repositories and compilation
sudo apt-get install build-essential cmake git ca-certificates curl

# Install required packages for OpenFOAM
sudo apt-get install flex libfl-dev bison zlib1g-dev libboost-system-dev libboost-
    thread-dev libopenmpi-dev openmpi-bin gnuplot libreadline-dev libncurses-dev
    libxt-dev
```

We will also install *ParaView* in order to visualize simulation results. ParaView depends on the *qt5-default* package, which is missing from the software repositories in the current release. However, there is a workaround. We will first install the *equivs* package. With this tool, the missing package can be created and all the required dependencies installed.

```
# Install the equivs package
sudo apt-get install equivs

# Switch to the "Downloads" directory
cd ~/Downloads

# Copy-paste below text (as is) to create the package information
cat <<EOF > qt5-default-control
Package: qt5-default
Source: qtbase-opensource-src
Version: 5.99.99
Architecture: all
Depends: qtbase5-dev, qtchooser
Suggests: qt5-qmake, qtbase5-dev-tools
Conflicts: qt4-default
Section: libdevel
Priority: optional
Homepage: http://qt-project.org/
Description: Qt 5 development defaults package
EOF

# Build and install the package
equivs-build qt5-default-control
sudo apt-get install ./qt5-default_5.99.99_all.deb

# Install other required packages
sudo apt-get install libqt5x11extras5-dev libxt-dev qttools5-dev ptscotch
```

We will now create the OpenFOAM installation directory and clone the source repository. Note the ~ character, which is a shortcut to denote your home directory. The **mkdir** command will thus create the OpenFOAM directory within your home directory. You will then switch to the new directory, using the **cd** command, and finally clone the packages from the remote repository, using the **git clone** command.

```
mkdir ~/OpenFOAM
cd ~/OpenFOAM
git clone https://github.com/OpenFOAM/OpenFOAM-10.git
git clone https://github.com/OpenFOAM/ThirdParty-10.git
```

By issuing the **ls** command, you can confirm that two new directories, `OpenFOAM-10` and `ThirdParty-10`, have been created. These directories contain the OpenFOAM source code and the source code of required third-party tools, including ParaView. To compile the OpenFOAM source code, set a permanent environment variable for OpenFOAM and add it to the source path.

```
echo 'source $HOME/OpenFOAM/OpenFOAM-10/etc/bashrc' >> ~/.bashrc
source $HOME/.bashrc
echo $WM_PROJECT_DIR
echo $ParaView_VERSION
```

Confirm that this has worked correctly. The second-last line should return the OpenFOAM directory (e.g. `/home/cfd/OpenFoam/OpenFOAM-10`) and the last line should return ParaView's version number (e.g. *5.6.3*). ParaView is the visualization application that comes bundled with OpenFOAM. It can be installed from the ThirdParty directory as follows.

```
cd ~/OpenFOAM/ThirdParty-10
./makeParaView
wmRefresh
```

The *makeParaView* script will now compile the ParaView source code. This means that the program's source code will be translated into machine code that can be executed on your computer. Depending on the allocated system resources (i.e. CPU, RAM), the compilation process may take a long time, up to several hours. Finish the process by typing the **wmRefresh** command, which will update the environment.

As the final step, we are going to compile OpenFOAM. Change to the OpenFOAM directory and run the *Allwmake* script. Again, please note that compilation may take several hours.

```
cd ~/OpenFOAM/OpenFOAM-10
./Allwmake -j
```

1.3 Simulation of a bubble column reactor

In this section, we will discuss the mixing behaviour inside a bubble column reactor. A bubble column reactor is a reactor type frequently encountered in different aerobic bioprocesses as it combines oxygen supply with in-situ mixing. Oxygen, or more typically air, is supplied from the reactor bottom such that the gas rises in the liquid column. The rising gas bubbles create a mixing effect.

Oxygen is required by the microorganisms not only for survival but also to allow the formation of desired biochemical products (e.g. antibiotics). At the same time, proper mixing is necessary to avoid local shortage or accumulation of substrate (e.g. oxygen, glucose). The lack or over-supply of the substrate can have detrimental effects on fermentation, lower product yield and productivity, undesired by-product formation, or even the formation of reactor dead zones due to starving cell populations. Consequently, the degree of mixing is an important design parameter in bioreactor engineering. CFD simulations are a powerful tool to provide deeper insights into the physico-chemical mechanisms occurring inside the reactor, and even inside the cells.

The reactor is a cylinder with height H = 1 m and internal diameter D = 0.2 m. In the absence of aeration, the initial height of the liquid in the reactor column is L =

0.6 m. The air stream at the inlet is pointing upwards and has a local velocity of v = 0.1 m/s. The reactor is operated at a reference pressure of p = 1 atm (101325 Pa) and the initial temperature is T = 30 °C (303.15 K) for both, the gas and the liquid. The problem geometry and operating parameters are summarized in Fig. 1.4.

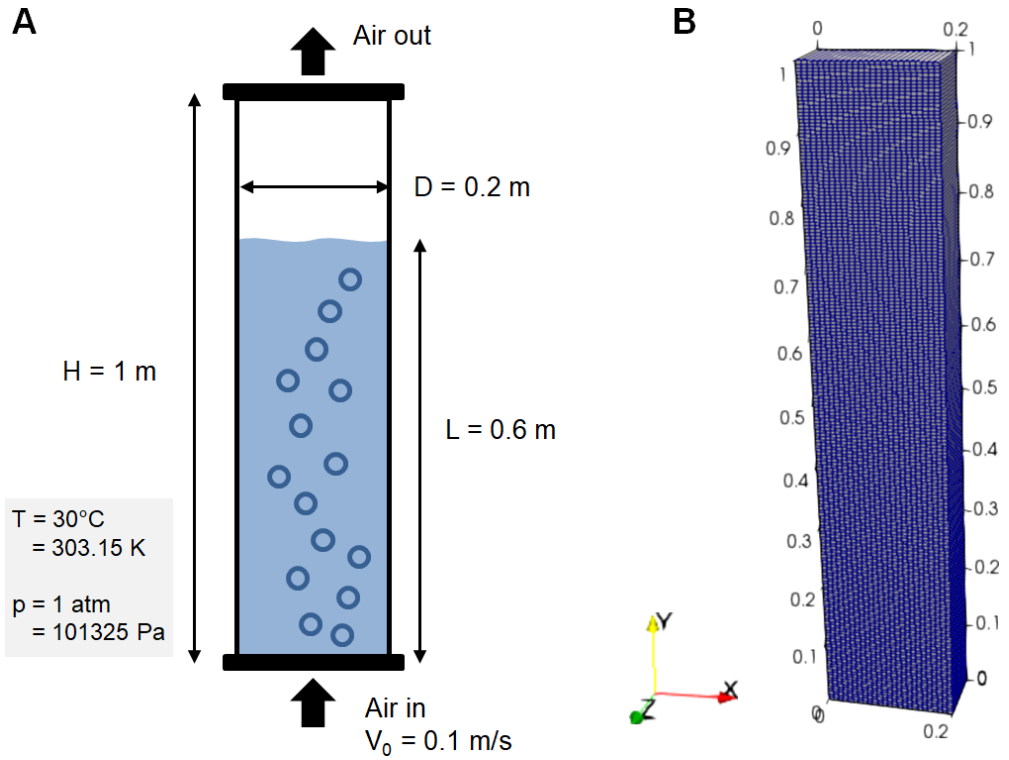


Figure 1.4 Problem specification for the simulation case: Subfigure A depicts the bubble column reactor with air supply from the bottom. The reactor has a height of H = 1 m and a diameter of D = 0.2 m. The reactor is initially filled with water up to a height of L = 0.6 m. Subfigure B depicts the mesh, representing the reactor configuration in a 2D plane through the reactor middle axis. The mesh measures 25 cells in the x-direction, 200 cells in the y-direction, and 1 cell in the z-direction.

We will now create a new OpenFOAM project and define the problem. We will then use OpenFOAM to calculate a solution to our problem definition, and finally, we will use ParaView to visualize the results. Bring up a terminal window and type the following commands to search for existing OpenFOAM templates relating to bubble column reactors. Select the *RAS/bubbleColumn* template and copy it to the *bioreactor* directory in your home directory.

```
cd $FOAM_TUTORIALS
find . -iname "*bubble*"
cp -r $FOAM_TUTORIALS/multiphase/multiphaseEulerFoam/RAS/bubbleColumn $HOME/
    bioreactor
```

You have now created a new project directory, named *bioreactor*, which is holding the simulation files. Switch to that directory by typing **cd /bioreactor**. Now, type the **ls** command to list the directory contents. *Folders* are indicated by bold blue letters, while *file names* are printed in white standard font. From here, you can inspect the individual project directories. For example, you should find the *constant*, *system*, and *0* directories. Each of these directories has a specific meaning within the simulation structure.

The *constant* directory specifies the physical properties for the simulation (e.g.

the properties of the individual phases such as water and air). The *system* directory contains parameters related to the solution procedure: The *controlDict* file specifies the simulation start/end time, the integration step size, and the numerical solver. In this tutorial, we will be using the **multiphaseEulerFoam** solver, which is a solver suited for a system of any number of compressible fluid phases with a common pressure, but otherwise separate properties. In other words, it is a solver that works well for the bubble column reactor, which contains a mixture of air and water. Additional solver settings and tolerances are described in the *fvSolutions* file. The problem geometry, or mesh, is described in the *blockMeshDict* file. Finally, the *0* directory is a special time directory that contains the initial conditions for the simulation. It will hold one text file for each field that is required for the particular solver (e.g. U for velocity, p for pressure). During the simulation, several new time directories will be created. The name of each time directory is based on the simulated time at which the data is written.

1.3.1 Mesh generation

Definition of the numerical mesh for the bubble column reactor can be found in the *blockMeshDict* file, located in the *system* directory. The bubble column reactor is defined as a vertical cylinder with a height of H = 1 m and a diameter of D = 0.2 m (compare Fig. 1.4). Nevertheless, for symmetry reasons, and to reduce the complexity of the simulation setup, we will consider this a 2D problem. We will therefore focus on the mixing behavior in a plane through the reactor middle axis (i.e. a cross-section of the reactor). The problem domain is therefore defined by eight vertices describing a single block of 0.2 m x 1.0 m x 0.1 m.

The block is discretized uniformly with 25 cells in the x-direction, 200 cells in the y-direction and 1 cell in the z-direction. The length assigned to the z-direction is arbitrary. With only a single cell in that direction, the problem geometry is effectively reduced to 2D, regardless of the length chosen. Three boundaries are defined: a gas inlet patch at the bottom, an outlet patch at the top, and two walls at the sides of the column. The boundaries are defined using the *faces* keyword, which refers to the defined vertices. For example, the inlet patch is framed by vertices 1, 5, 4, and 0.

```
convertToMeters 1;

vertices
(
    (0    0  0  )  // 0
    (0.2  0  0  )  // 1
    (0.2  1  0  )  // 2
    (0    1  0  )  // 3
    (0    0  0.1)  // 4
    (0.2  0  0.1)  // 5
    (0.2  1  0.1)  // 6
    (0    1  0.1)  // 7
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (25 200 1) simpleGrading (1 1 1)
);

defaultPatch
{
    type empty;
```

```
}

boundary
(
    inlet
    {
        type patch;
        faces
        (
            (1 5 4 0)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (3 7 6 2)
        );
    }
    walls
    {
        type wall;
        faces
        (
            (0 4 7 3)
            (2 6 5 1)
        );
    }
);
```

You can open and modify the *blockMeshDict* file directly in the terminal or by running **mouspad blockMeshDict &** from within the *system* directory. Once you have finished editing the *blockMeshDict*, it is time to generate the mesh. In the terminal window, navigate to your project main directory and run the following commands:

```
# change to the project main directory
cd ~/bioreactor

# build and inspect the mesh
blockMesh
checkMesh
paraFoam
```

The **blockMesh** command will generate the mesh and provide some basic mesh statistics. By running the **checkMesh** command, you can obtain more detailed information on mesh quality parameters. You can inspect the generated mesh visually by running the **paraFoam** command, which is an OpenFOAM-specific wrapper for ParaView. Mmesh example is given in Fig. 1.4.

1.3.2 Physical properties and phases

We will now include the physics in the simulation setup. We will begin with the definition of two discrete phases, **water** and **air**. These are defined in the *constant/phaseProperties* file, using the *phases* keyword. The *air* keyword holds definitions for the **diameterModel**, **isothermalCoeffs** and **residualAlpha**. The isothermal diameter model has been selected to describe the behavior of gas bubbles in the system. This

model describes the change in gas bubble diameter as a function of pressure. The d0 and p0 parameters set the reference conditions for the gas bubbles, namely a mean bubble diameter of 3 mm at a reference pressure of 10^5 Pa. In large bubble column reactors, this model accounts for the fact that gas bubbles are small at the bottom of the reactor when they are formed. But as the bubble rises to the surface, its diameter increases (bubble growth). In contrast, the *water* keyword defines a constant diameter model for water droplets with a mean diameter of 0.1 mm.

The *blending* method sets conditions on the mixing behaviour of the two phases, so as to instruct the solver when the phases should be considered as dispersed, mixed or continuous. Under *default*, a linear blending is assumed, which makes this model applicable to all phase interactions except *drag*, which is defined in a separate dictionary below. The logic for the defined conditions is as follows; the parameter **minFullyContinuousAlpha.air** specifies the minimum volume fraction of air to be considered as a continuous phase. Once this volume fraction is surpassed, air will be considered as a continuum and the other phase will become the dispersed phase. The parameter **minPartlyContinuousAlpha.air** specifies the minimum volume fraction of the gas phase in a cell volume which can be considered as a dispersed phase. Above this value, air can be considered mixed with water, but below that threshold, air will be considered as the dispersed phase (i.e. gas bubbles). The parameters are defined analogously for water.

A constant surface tension of 0.07 N/m is set for the *surfaceTension* keyword. This definition is applied by the solver whenever air is dispersed in water. The *drag* keyword defines models for inter-phase momentum transfer. For dispersed phases, either water in the air, or air in water, the *SchillerNaumann* model is used. In the absence of dispersed phases, a segregated drag model is used. The next dictionary defines a *virtualMass* model for water dispersed in air, and for air dispersed in water. A virtual mass is introduced in the momentum equations to describe the required acceleration force on the mass of the surrounding continuous phase, if a dispersed phase fragment, such as a bubble or droplet, changes its velocity relative to the surrounding phase. Here, a *constantCoefficient* model is used with a virtual mass coefficient (Cvm) equal to 0.5. Finally, the *heatTransfer* keyword describes heat transfer between air bubbles dispersed in water as well as water droplets dispersed in the air. For this purpose, the *RanzMarshall* model has been selected.

```
type    basicMultiphaseSystem;

phases  (air water);

air
{
    type            purePhaseModel;
    diameterModel   isothermal;
    isothermalCoeffs
    {
        d0          3e-3;
        p0          1e5;
    }
    residualAlpha   1e-6;
}

water
{
    type            purePhaseModel;
    diameterModel   constant;
```

```
    constantCoeffs
    {
        d           1e-4;
    }
    residualAlpha   1e-6;
}

blending
{
    default
    {
        type linear;
        minFullyContinuousAlpha.air 0.7;
        minPartlyContinuousAlpha.air 0.3;
        minFullyContinuousAlpha.water 0.7;
        minPartlyContinuousAlpha.water 0.3;
    }
    drag
    {
        type linear;
        minFullyContinuousAlpha.air 0.7;
        minPartlyContinuousAlpha.air 0.5;
        minFullyContinuousAlpha.water 0.7;
        minPartlyContinuousAlpha.water 0.5;
    }
}

surfaceTension
{
    air_water
    {
        type            constant;
        sigma           0.07;
    }
}

drag
{
    air_dispersedIn_water
    {
        type            SchillerNaumann;
        residualRe      1e-3;
    }
    water_dispersedIn_air
    {
        type            SchillerNaumann;
        residualRe      1e-3;
    }
    air_segregatedWith_water
    {
        type            segregated;
        m               0.5;
        n               8;
    }
}

virtualMass
{
    air_dispersedIn_water
    {
        type            constantCoefficient;
```

```
        Cvm             0.5;
    }
    water_dispersedIn_air
    {
        type            constantCoefficient;
        Cvm             0.5;
    }
}

heatTransfer
{
    air_dispersedIn_water
    {
        type            RanzMarshall;
        residualAlpha   1e-4;
    }
    water_dispersedIn_air
    {
        type            RanzMarshall;
        residualAlpha   1e-4;
    }
}
```

1.3.3 Turbulence model

The turbulence properties for the two phases, water and air, are defined in the *constant/momentumTransport.water* and *constant/momentumTransport.air* files, respectively. For both phases, we will apply the mixtureKEpsilon turbulence model with default model coefficients.

```
simulationType  RAS;

RAS
{
    model           mixtureKEpsilon;
    turbulence      on;
    printCoeffs     on;
}
```

1.3.4 Boundary and initial conditions

Initial and boundary conditions are defined in the *0* directory. Use the editor to inspect the contents of the individual files, representing different fields. For example, **U.air** and **U.water** represent the velocity field of the two phases, while **T.air** and **T.water** corresponds to the temperature field. The files **alpha.air** and **alpha.water** refer to the volume fraction of either phase. In the first step, we will set the desired inlet gas velocity of v = 0.1 m/s. Open the *0/U.air* file and modify according to the problem specifications to give:

```
dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0.1 0);

boundaryField
{
    inlet
    {
```

```
        type            fixedValue;
        value           $internalField;
    }
    outlet
    {
        type            pressureInletOutletVelocity;
        phi             phi.air;
        value           $internalField;
    }
    walls
    {
        type            fixedValue;
        value           uniform (0 0 0);
    }
}
```

Open the *0/T.air* and *0/T.water* dictionaries and set the initial temperature to 303.15 K. The modified file for *T.air* should contain the following:

```
dimensions          [0 0 0 1 0 0 0];

internalField       uniform 303.15;

boundaryField
{
    walls
    {
        type            zeroGradient;
    }
    outlet
    {
        type            inletOutlet;
        phi             phi.air;
        inletValue      $internalField;
        value           $internalField;
    }
    inlet
    {
        type            fixedValue;
        value           $internalField;
    }
    frontAndBackPlanes
    {
        type            empty;
    }
}
```

Analogously, *T.water* dictionary should be defined as follows:

```
dimensions          [0 0 0 1 0 0 0];

internalField       uniform 303.15;

boundaryField
{
    walls
    {
        type            zeroGradient;
    }
    outlet
```

```
    {
        type            inletOutlet;
        phi             phi.water;
        inletValue      $internalField;
        value           $internalField;
    }
    inlet
    {
        type            fixedValue;
        value           $internalField;
    }
    frontAndBackPlanes
    {
        type            empty;
    }
}
```

The reactor column is initially filled with water, up to a height of 0.6 m. This condition is described in the *system/setFieldsDict* dictionary. Edit the file to include the settings below. First, the *defaultFieldValues* are applied, which fills the entire column with air. Then, the values specified in the *regions* entry will overwrite the defaults where applicable. We define a box with two points (0 0 0) and (0.2 0.6 0.1), and specify for all cells in that box that the volume shall be filled with water. Following the modifications, execute the **setFields** command from your main project directory. The initial conditions should now take effect.

```
defaultFieldValues
(
    volScalarFieldValue alpha.air 1
    volScalarFieldValue alpha.water 0
);

regions
(
    boxToCell
    {
        box (0 0 0) (0.2 0.6 0.1);
        fieldValues
        (
            volScalarFieldValue alpha.air 0
            volScalarFieldValue alpha.water 1
        );
    }
);
```

1.3.5 Solver settings

As mentioned earlier, the *controlDict* dictionary in the *system* directory specifies important solver settings, such as the simulation end time, the integration step size, and the numerical solver to be used. We want to use the **multiphaseEulerFoam** solver and this setting should already be set. Open the *controlDict* file in an editor to set the simulation **endTime** to 60 s. This will be sufficient to obtain a general overview of the process. Make sure that the **writeInterval** is set to 1 s. This means that the simulation results are written to the file every second until reaching **endTime**. Save the file when you are finished editing to obtain the following:

```
application     multiphaseEulerFoam;
startFrom       startTime;
startTime       0;
stopAt          endTime;
endTime         60;
deltaT          0.005;
writeControl    runTime;
writeInterval   1;
purgeWrite      0;
writeFormat     ascii;
writePrecision  6;
writeCompression off;
timeFormat      general;
timePrecision   6;
runTimeModifiable yes;
adjustTimeStep  no;
maxCo           0.5;
maxDeltaT       1;
```

1.3.6 Simulation results

To start the simulation, execute the following commands in a terminal window. Depending on the computational power of your system, this process will take several minutes (perhaps an hour or more) to complete. Already from the run-time of this rather simplistic modelling case, you can get an idea of why HPC plays an important role in more detailed (e.g. finer mesh, more cells, 3D geometry) and more complex (e.g. variable feeding regimes, inclusion of biokinetics) CFD simulations.

```
# change to the project main directory
cd ~/bioreactor

# Prepare and execute the simulation
blockMesh
setFields
multiphaseEulerFoam

# Visualize the results
ParaFoam
```

Once the solver has finished, you can visualize the results using ParaFoam. Figures 1.5-1.10 provide an overview of the flow pattern and phase distribution at different time steps counting from reactor startup. At t = 0 s, the reactor is filled with water up to a height of 0.6 m (alpha.air = 0; alpha.water = 1) with a continuum of air above (alpha.air = 1; alpha.water = 0). The velocity field (U.air) indicates zero flow at this point. At t = 2 s, air inflow from the bottom is evident and, as a result of the increased gas holdup, the gas-liquid interface is moving higher up in the column. At t = 5 s, the flow pattern becomes more turbulent and the gas holdup keeps increasing. Between t = 10 s and t = 60 s, the air flow develops a more recognizable, stable pattern. Nevertheless, the flow remains turbulent and there is a channeling effect of the air stream passing through the reactor. This is marked by zones of relatively high gas velocity.

Using the output written in the main directory, i.e. the time directories created during simulation, it is possible to perform more accurate data analysis, such as calculating the local gas holdup and therefore to improve reactor design and operation. However, any simulation should be implemented with care. Experimental validation and simulation should go hand in hand to ensure the validity of obtained results.

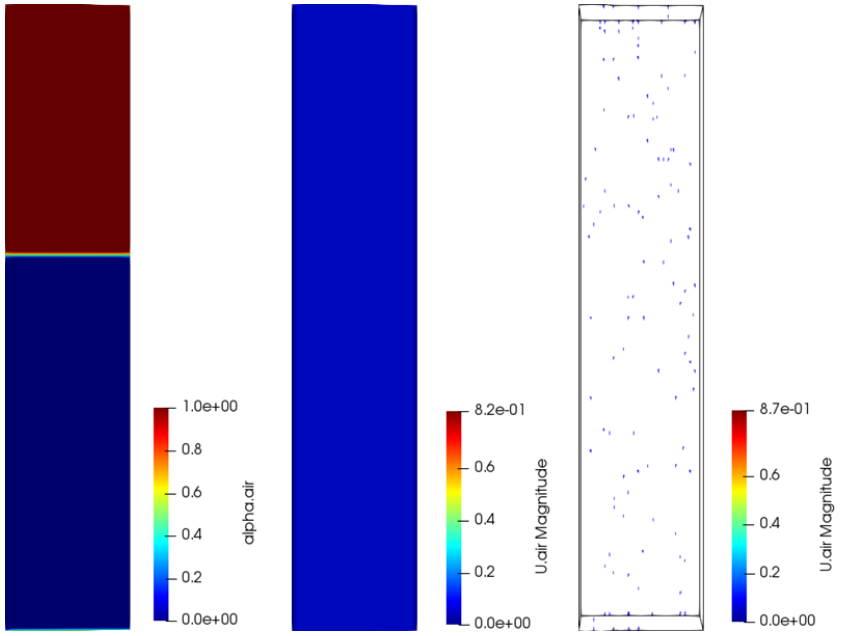


Figure 1.5 Phase distribution and air velocity profiles at t = 0 s.

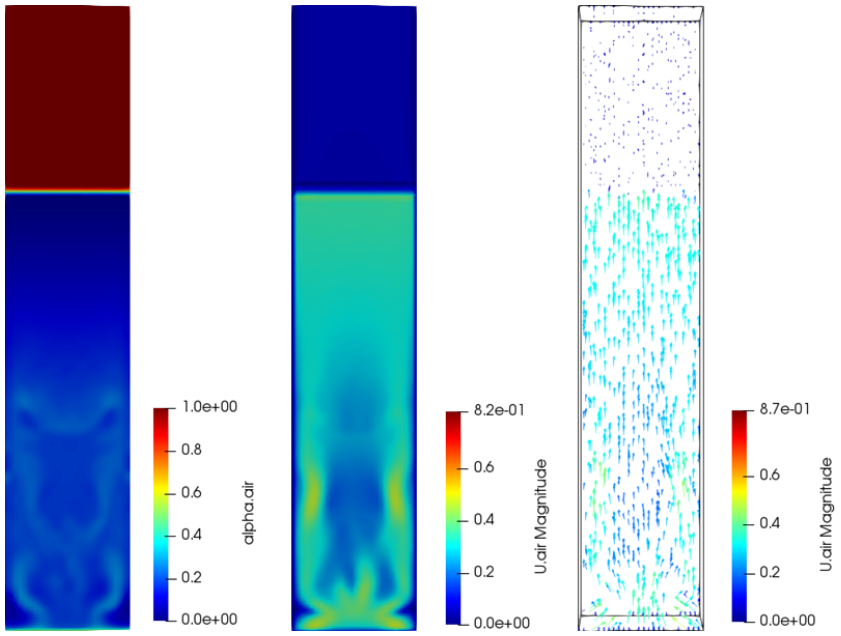


Figure 1.6 Phase distribution and air velocity profiles obtained at t = 2 s.

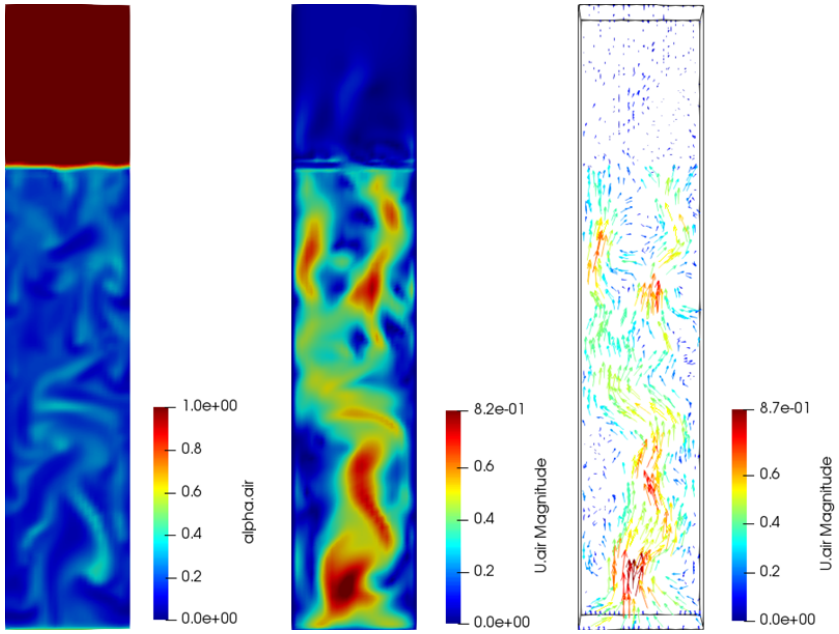


Figure 1.7 Phase distribution and air velocity profiles obtained at t = 5 s.

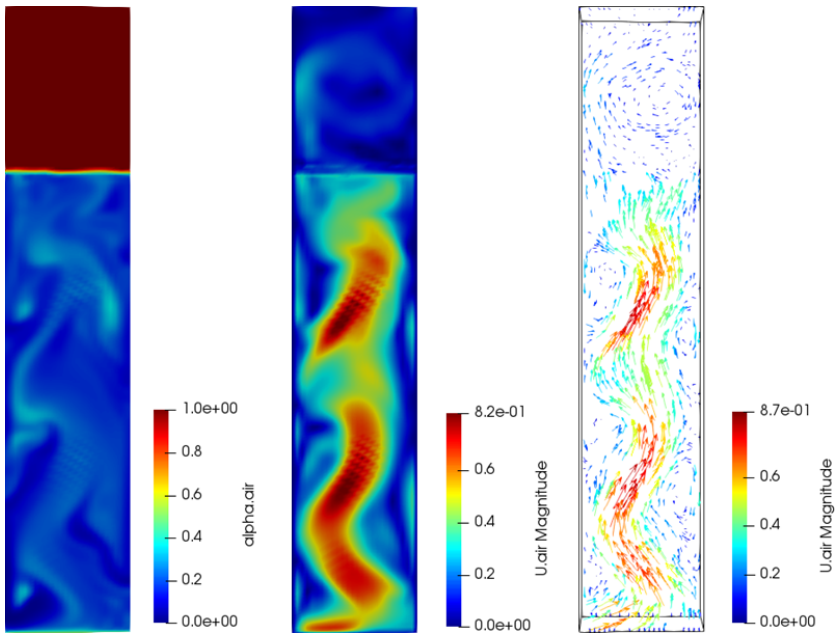


Figure 1.8 Phase distribution and air velocity profiles obtained at t = 10 s.

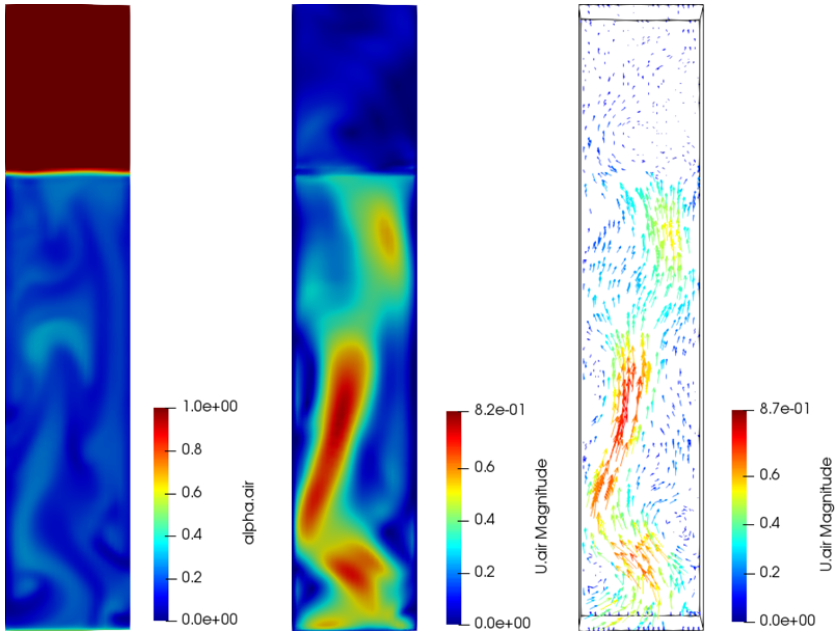


Figure 1.9 Phase distribution and air velocity profiles obtained at t = 30 s.

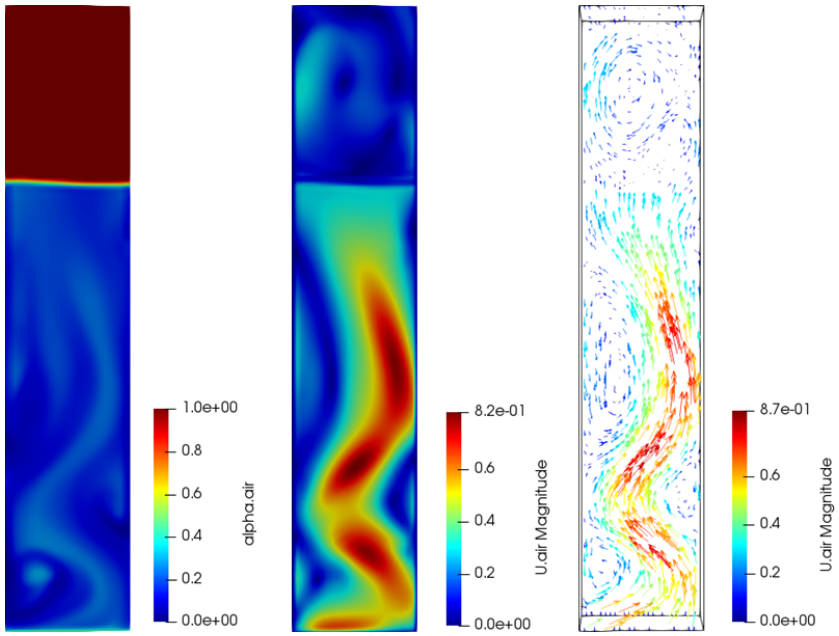


Figure 1.10 Phase distribution and air velocity profiles obtained at t = 60 s.

1.4 Simulation of complex fluid dynamic fields

University of Trieste

In the following chapters, we will discuss the applicability of OpenFOAM for non-standard fluid dynamic cases. Specifically, we will consider two different types of problems:

- Rayleigh-Bénard convection in a cylindrical cell at high Ra numbers
- water wave loads on a rectangular floating body.

1.4.1 Rayleigh-Bénard convection in a cylindrical cell

We will first discuss the case of Rayleigh-Bénard (RB) convection. In RB convection, a thermo-fluid dynamic field is driven by buoyancy effects. The typical case is when the fluid is confined between two horizontal plates, with the top one at a temperature lower than the bottom one. The temperature gap causes the variation in the density of the fluid, with a larger density on the top, and induces vertical motion and mixing. This is a fundamental mixing process that occurs in the environment, both in the atmosphere and in water basins.

Typically, this class of problems is studied under the assumption that the density variations are small compared to the bulk density of the fluid. Furthermore, it is considered that the advective accelerations are small compared to gravity and that the vertical scale of motion is small compared to the adiabatic lapse (Boussinesq approximation). Under the Boussinesq approximation, the momentum and thermal diffusivity are considered independent of temperature. The conservation equations assume the following form:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{1.1}$$

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{\partial P}{\partial x_i} + g\alpha[T - T_0]\delta_{i3} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} \tag{1.2}$$

$$\frac{\partial T}{\partial t} + u_i \frac{\partial T}{\partial x_i} = k\nabla^2 T \tag{1.3}$$

where u_i is the velocity component along the x_i-direction, P is the kinematic pressure, g is the gravitational acceleration, α is the cubic expansion coefficient, T and T_0 are respectively the temperature field and the reference temperature and ν is the kinematic viscosity of the fluid. $k = \nu/Pr$ with Pr as the Prandtl number is the thermal diffusivity. The set of equations takes advantage of the state equation which, under the Boussinesq approximation, reads as:

$$\frac{\Delta\rho}{\rho_0} = -\alpha\Delta T \tag{1.4}$$

where $\Delta T = T - T_0$ and $\Delta\rho = \rho - \rho_0$ where ρ and ρ_0 are the density of the fluid at the temperature T and T_0 respectively.

We consider a cylindrical cell of aspect ratio $\Gamma = D/H = 1/2$ heated from below and cooled from above, with adiabatic sidewalls. The physical problem is controlled by three non-dimensional parameters:

$$Ra = \frac{\alpha\Delta T H^3 g}{\nu k}, \quad Pr = \frac{\nu}{k}, \quad \Gamma$$

which are respectively the Rayleigh number, the Prandtl number and the aspect ratio of the cell.

We consider a turbulent problem and the equations are solved according to the filtered approach. Specifically, the variables are filtered using a low-pass filter where the subgrid stress needs closure. This approach, called Large Eddy Simulation, is able to reproduce directly the large anisotropic and energy-carrying structures of the flow, whereas, the small, more universal and dissipative scales of turbulence need parametrization. The subgrid-scale stresses are here parametrized using the WALE model, whereas, for the SGS buoyancy fluxes we use $Pr_{sgs} = 1.0$

The numerical simulations are carried out for $Pr = 0.7\boldsymbol{\nabla} \cdot 1$ and two different Rayleigh numbers, $2 \cdot 10^7$ and $2 \cdot 10^{10}$ respectively. We use *wall-resolving* LES, meaning that we solve the turbulent field together with the thermal and the momentum boundary layers up to the wall, therefore, over the solid surfaces we set no-slip conditions. A discussion on the resolution requirements is necessary in order to assess the quality of the results. The results obtained in the case of $Ra = 2 \cdot 10^7$ are compared with those of Verzicco and Camussi [Verzicco and Camussi, 2003] who performed DNS in a cylindrical cell for Pr$= 0.7$ and $\Gamma = 1/2$. The DNS results are also used to evaluate the reliability of our simulation in the case of Ra$= 2 \cdot 10^{10}$.

The large number of dimensional parameters can be reduced by identifying, respectively, a length, a velocity and a temperature scale in the system. The most common way is to take the height H of the domain as length scale L, and the free-fall velocity U_f, defined as $U_f \equiv \sqrt{g\alpha(T - T_0)H}$ as velocity scale. Further, the temperature is made non-dimensional in the following way:

$$\begin{aligned} \theta &= \frac{T - T_0}{T_{hot} - T_{cold}} \\ T_0 &= T_{hot} \end{aligned} \tag{1.5}$$

where T_{hot} is the temperature of the lower hotter plate and T_{cold} is the temperature of the colder, upper plate. Hence, θ is 0 and 1 at the cold plate and hot plate, respectively.

We have used the open-source CFD software OpenFOAM v2206. OpenFOAM (Open-source Field Operation And Manipulation) is an object-oriented $C++$ framework that can be used to build a variety of computational solvers for problems in continuum mechanics, with a focus on finite volume discretization. At the core of these libraries, there is a set of object classes that allow the programmer to manipulate meshes, geometries, and discretization techniques at a high level of coding. For the specific problem of RB convection, we shall use two different solvers. The solver *buoyantBoussinesqPisoFoam* with a second-order central discretization in space and a second-order discretization in time. This is an unsteady solver for buoyant, incompressible turbulent flows customized to obtain variables useful for the subsequent analysis of the data and to enable the LES capabilities of the solver permitting the use of different turbulence models. The other solver is *buoyantBoussinesqRungeKuttaFoam*, an energy-conserving incremental-pressure written for [López Castaño et al., 2019] and implemented in the new OpenFOAM version for our purposes. It is a third-order accurate method because the non-solenoidal velocity obtained from the momentum equation is corrected three times. As demonstrated in our simulation results, the RK4 algorithm together with the WALE model gives better results for Rayleigh-Bénard convection for both first- and second-order statistics. The superiority of RK4 over PISO is a consequence of the dissipative features exhibited by the latter. Additionally, the results for PISO improve with increasing value of the Ra number. The code solves

the governing equations using a non-staggered grid mesh, where the pressure and Cartesian velocity components are defined at the center of the grid whereas the volume fluxes are defined at the midpoint of their corresponding faces of the cell. The stability of the overall numerical method is limited by the Courant-Friedrichs-Lewy (CFL) condition. The local CFL number is defined as:

$$CFL = \left(\frac{|u_1|}{\Delta x} + \frac{|u_2|}{\Delta y} + \frac{|u_3|}{\Delta z}\right)\Delta t = (|U_1| + |U_2| + |U_3|)\frac{\Delta t}{J^{-1}} \tag{1.6}$$

where $\Delta x, \Delta y, \Delta z$ are the grid spacings over the three Cartesian coordinates. In the above fractional steps, the stability condition requires that the maximum value obtained from equation (1.6) in the computational domain is:

$$CFL_{max} < \bar{C} \sim 1. \tag{1.7}$$

The $\bar{C}$ is a function of the Reynolds number and it may become smaller than one for a highly skewed grid mesh. In this application, the time step is not adaptive and the CFL_{max} has been set as 0.5. The table 1.1 sums up the number of processors and the time for each iteration in every case we simulate.

Table 1.1 Number of processors used and computational time for each iteration in all simulations.

	Ra	n CPU	Iteration time
LES7PISO and LES7RK	$2 \cdot 10^7$	14	2 s
LES10PISO and LES10RK	$2 \cdot 10^{10}$	16	1.45 s

The mesh is generated in such a way as to solve the boundary layers. This means that within the momentum and thermal boundary layers, δ_u and δ_θ respectively, it is necessary to place a minimum number of grid cells. In the bulk region, the mesh size in the vertical direction is determined as ten times the Kolmogorov scale. According to [Verzicco and Camussi, 2003] a reasonable estimate of the Kolmogorov scale is:

$$\frac{\eta}{h} \simeq \pi \left(\frac{Pr^2}{Ra\,Nu}\right)^{1/4}. \tag{1.8}$$

The thermal boundary layer is very thin at horizontal plates where the temperature is assigned as BC. On the other hand, the momentum boundary layer is thin at the vertical adiabatic walls, where energetic ascendent/descendent flow is present. We use a nonuniform grid in the radial as well as in the vertical direction, whereas grid spacing is constant along the circumferential direction. According to [Verzicco and Camussi, 2003], we can evaluate the thermal boundary layer thickness as smaller than the momentum one ($\delta_\theta < \delta_u$). The former can be reasonably evaluated according to:

$$\delta_\theta \simeq \frac{h}{2\,Nu}. \tag{1.9}$$

This estimate for δ_θ is used to define the number of grid points to be placed in the boundary layers. In our case, we use a minimum of six cells within the thermal boundary layer with the first one located around $\delta_\theta/8$ from the wall. In the bulk region, in order to reduce the computational costs, we use a grid in the vertical direction ten times the Kolmogorov scale η. This choice is made in accordance with what was reported in [Pope, 2000]. It was surmized that η underestimates the size

of the dissipative motions and Monin [Monin and Yaglom, 1975] concluded that the separation of the inertial range from the dissipative one occurs at about 10η ([Monin and Yaglom, 1975], [Verzicco and Camussi, 2003]). For the radial grading, we use the same cell-to-cell expansion ratio as for the vertical direction, with the first cell near the sidewall of size $\delta_\theta/8$. However, in the angular direction, we use a uniform grid size that is 50 times that of the smallest one within the boundary layer.

The resolution of the grid for the whole simulations performed for Ra$= 2 \cdot 10^7$ is listed in the table 1.2. Moreover, it includes the resolution of the grid for the DNS simulation conducted in [Verzicco and Camussi, 2003]. Using the same approach, in the table 1.3 we report the grid resolutions for the Ra$= 2 \cdot 10^{10}$ cases. We indicate with LES7PISO and LES7PISO1 the results obtained with the PISO algorithm for Ra$= 2 \cdot 10^7$ and for Pr$= 0.7$ and Pr$= 1.0$, respectively. With LES7PISOF we indicate the Ra$= 2 \cdot 10^7$ and Pr$= 0.7$ case but with a finer grid in the azimuthal direction. LES7RK and LES7RK1 represent the same case but with the RK4 discretization method. Finally, with DNS7 we indicate the DNS results of [Verzicco and Camussi, 2003].

Table 1.2 Physical and computational parameters for Rayleigh number $2 \cdot 10^7$.

	Ra	**Pr**	η/h	10η	δ_θ	$\delta_\theta/8$	$N_\theta \times N_r \times N_z$
LES7PISO	$2 \cdot 10^7$	0.7	0.02160	0.432	0.04473	0.005591	$65 \times 49 \times 99$
LES7PISO1	$2 \cdot 10^7$	1.0	0.02160	0.432	0.04473	0.005591	$65 \times 49 \times 99$
LES7PISOF	$2 \cdot 10^7$	0.7	0.02160	0.432	0.04473	0.005591	$97 \times 49 \times 99$
LES7RK	$2 \cdot 10^7$	0.7	0.02160	0.432	0.04473	0.005591	$65 \times 49 \times 99$
LES7RK1	$2 \cdot 10^7$	1.0	0.02160	0.432	0.04473	0.005591	$65 \times 49 \times 99$
DNS7	$2 \cdot 10^7$	0.7	0.018	0.0076	0.0446	0.005575	$97 \times 49 \times 193$

For the cases with Ra$= 2 \cdot 10^{10}$ we indicate with LES10RK the case with Pr$= 0.7$ and the RK4 as discretization algorithm, and with LES10RK1 the case with Pr$= 1.0$. LES10PISO1 indicates the simulation carried out with the PISO algorithm and Pr$= 1.0$ and DNS10 the results of [Verzicco and Camussi, 2003].

Table 1.3 Physical and computational parameters for Rayleigh number $2 \cdot 10^{10}$.

	Ra	**Pr**	η/h	10η	δ_θ	$\delta_\theta/8$	$N_\theta \times N_r \times N_z$
LES10PISO1	$2 \cdot 10^{10}$	1.0	0.002253	0.04506	0.00529	0.000661	$96 \times 75 \times 188$
LES10RK	$2 \cdot 10^{10}$	0.7	0.002253	0.04506	0.00529	0.000661	$96 \times 75 \times 188$
LES10RK1	$2 \cdot 10^{10}$	1.0	0.002253	0.04506	0.00529	0.000661	$96 \times 75 \times 188$
DNS10	$2 \cdot 10^{10}$	0.7	0.0018	0.0048	0.0013	0.0001625	$129 \times 97 \times 385$

Verzicco and Camussi [Verzicco and Camussi, 2003] provide a criterion to evaluate the duration of each simulation since analogous statistics for turbulent quantities had to be computed. This duration is defined in terms of large-eddy-turnover times T_L assuming a fluid particle to revolve inside the cell at a speed of the order of the free-fall velocity along an elliptic path [Verzicco and Camussi, 2003]. This T_L time increases with the Ra number and only after a large number of large-eddy-turnover times all the statistical quantities averaged can be considered converged. If $U = \sqrt{g\alpha\Delta T h}$ we can estimate $T_L \simeq 2h/U$ and for $Ra = 2 \cdot 10^7$ the criterion gives $T_{tot} = 100\,T_L$, while for $Ra = 2 \cdot 10^{10}$ gives $T_{tot} = 165\,T_L$. For $Ra = 2 \cdot 10^{12}$, since it is not reported in [Verzicco

and Camussi, 2003], we consider the same T_{tot} reported for the range $2 \cdot 10^{10} - 2 \cdot 10^{11}$, i.e. $T_{tot} = 275\, T_L$. This refers to the calculation of the time averages of the variables after bringing the field to a stage that can be defined as fully developed. All the references for the case are reported in tables 1.4 and 1.5 for Ra$= 2 \cdot 10^7$ and Ra$= 2 \cdot 10^{10}$ respectively, where T_{stab} indicates the range of time we use to bring the flow structure in a fully developed state.

Table 1.4 Duration of simulation as a function of large-eddy-turnover time for Ra$= 2 \cdot 10^7$.

	Ra	U	T_L	$100 \cdot T_L$	T_{tot}	T_{stab}
LES7PISO	$2 \cdot 10^7$	$\sqrt{2}$	$2\sqrt{2}$	282	280	120
LES7PISO1	$2 \cdot 10^7$	$\sqrt{2}$	$2\sqrt{2}$	282	280	145
LES7PISOF	$2 \cdot 10^7$	$\sqrt{2}$	$2\sqrt{2}$	282	325	354
LES7RK	$2 \cdot 10^7$	$\sqrt{2}$	$2\sqrt{2}$	282	395	200
LES7RK1	$2 \cdot 10^7$	$\sqrt{2}$	$2\sqrt{2}$	282	390	200

Table 1.5 Duration of simulation as a function of large-eddy-turnover time for Ra$= 2 \cdot 10^{10}$.

	Ra	U	T_L	$165 \cdot T_L$	T_{tot}	T_{stab}
LES10PISO1	$2 \cdot 10^{10}$	$\sqrt{2}$	$2\sqrt{2}$	467	490	163
LES10RK	$2 \cdot 10^{10}$	$\sqrt{2}$	$2\sqrt{2}$	467	490	410
LES10RK1	$2 \cdot 10^{10}$	$\sqrt{2}$	$2\sqrt{2}$	467	490	410

The results obtained for Ra $= 2 \cdot 10^7$ and Pr$= 0.7$ using the PISO and RK4 solvers are reported below and compared to those of [Verzicco and Camussi, 2003]. It is also possible to compare the statistics obtained with PISO and RK4 algorithms. First of all, we turn our attention to the calculation of the Nusselt number. Two definitions were used to calculate the Nusselt number, the first named Nu_{plate} and defined by equation (1.10), the second named Nu_{int} and defined by the equation (1.11). The former is the Nusselt number obtained from the gradient of the mean temperature evaluated on the bottom and top plates:

$$Nu_{plate} = \left. \frac{\partial \langle \theta \rangle_{t,s}}{\partial z} \right|_{z=0,1} \tag{1.10}$$

where $\langle \theta \rangle_{t,s}$ is the average over time and in the horizontal plane at $z = 0$ and $z = 1$, which correspond to the lower and upper plate. This value defines an indirect way to check the grid resolution in the thermal boundary layer. Nu_{int} is defined as:

$$Nu_{int} = 1 + \sqrt{Ra\, Pr}\, \langle w\theta \rangle_{t,V} \tag{1.11}$$

where $\langle w\theta \rangle_{t,V}$ is the mean over time and the whole domain of the product between the vertical component of velocity and the temperature. Nu_{int} can be used as an indicator of the grid resolution in the bulk region of the domain to determine whether the dynamic of the mean flow is represented. These two values are evaluated at every time step and then averaged in time. Simulation results are compared to those of [Verzicco and Camussi, 2003] (DNS7) in the table 1.6.

The values shown in table 1.6 require additional explanation. First of all, we have to point out that the error for the Nu has been calculated as the third-order standard

Table 1.6 Value of Nu_{int} for three considered simulations.

	Nu_{int}
LES7PISO	20.11 ± 0.66
LES7RK	22.57 ± 1.02
DNS7	20.56 ± 1.48

deviation on a discrete set of data generated every five seconds of the simulation and the value of Nu is averaged in time. The Nu_{int} evaluated using both the PISO and RK4 algorithms is comparable with the results presented in [Verzicco and Camussi, 2003]. In particular, the LES7RK value is higher than the LES7PISO one.

The mean flow structure can be partially represented by showing the vertical profiles of the mean vertical velocity and temperature at the axis of the cylindrical cell. Figure 1.11 shows the mean vertical velocity component at the axis ($r = 0$) of the bottom half of the domain. In all the figures the vertical distance from the bottom plate y is normalized with the height of the cell h. We can see that for $\mathrm{Ra} = 2 \cdot 10^7$ the vertical velocity at the axis is negative for $y/h \leq 0.5$ and positive for $y/h > 0.5$. This is a clear sign of the presence of two counter-rotating toroidal rings attached to the horizontal plates.

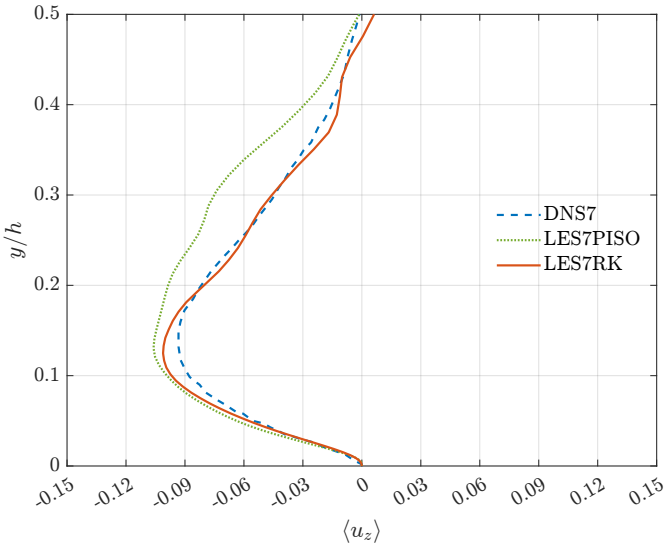


Figure 1.11 Mean vertical profiles of the vertical velocity $\langle u_z \rangle$ at the axis of the cell. The dashed line represents the DNS results. The solid line represents LES7RK results and the dotted line LES7PISO results.

From figure 1.11 we can see how within the boundary layer ($\delta_\theta \sim 0.049$) both RK4 and PISO algorithms give consistent results with the DNS. The main differences can be viewed in the bulk region. The LES7RK results are nearly identical to the DNS with a small overestimation of the peak value at $y/h \sim 0.12$, while LES7PISO provides a mean velocity profile that does not sufficiently represent the physics of the problem.

For the cylindrical domain, the dynamic in the central region of the cell can be described with a large-scale circulation (LSC) where the fluid at the hot plate rises near the lateral wall and descends at the center of the cell. This motion of the fluid can be viewed in figure 1.12 where the streamlines of the vertical velocity u_z in the entire domain are shown. Red lines indicate the streamlines of the positive component of u_z

that from the hot bottom plate rise in the cell center. Inside the cylinder we can see the descending component of the vertical velocity in blue.

Figure 1.12 Streamlines representing the instantaneous vertical velocity component in the entire volume of the cylinder in the case of Ra$= 2 \cdot 10^7$.

Near the upper and bottom plate the flow is dominated by axisymmetric torodial rings. These particular structures are displayed in figure 1.13 where the streamlines of the vertical velocity instantaneous field are plotted in the hot plate.

Figure 1.13 Streamlines representing the instantaneous vertical velocity component in the hot plate at the bottom of the cylinder in the case of Ra$= 2 \cdot 10^7$.

From the visualization of the instantaneous field at a specific simulation time, no information about the dynamics of the flow can be deduced. However, we can assess whether our results can describe the flow configuration shown in figures 1.12 and 1.13. For this purpose, we compute the local friction coefficients as the normalized wall-normal derivative of the tangential velocity. The normalization factor is $1/Re = \sqrt{Pr/Ra}$. In particular, in figure 1.14 the radial profile is obtained by plotting the tangential velocity as the projection of the velocity vector onto the plane parallel to the plate. The LES4RK profile is able to represent the DNS one in particular near the sidewall of the domain. The maximum value is located at the same radius, while in the LES1PISO case is overestimated and occurs at a greater radius value. Near the axis of the cylindrical cell $(r/d = 0)$ the values of $(1/Re)(\delta u_t/\delta n)$ in both simulations have a trend that forms that of DNS and this may be due to insufficient refinement of the grid near the center.

For the vertical profile, the tangential velocity is the projection of the velocity vector onto the plane parallel to the lateral sidewall and it is averaged in time and the azimuthal direction. Figure 1.15 shows that our simulations are able to reproduce the

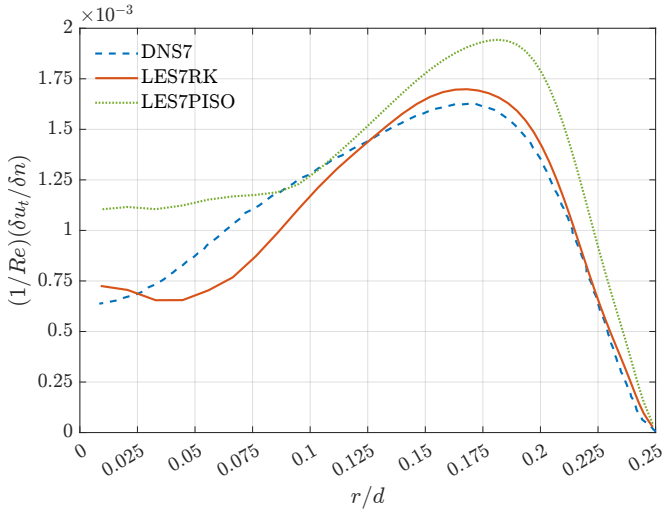


Figure 1.14 Radial profiles at the horizontal plate of the mean (the profiles are averaged in time and between the lower and upper plate) wall-normalized velocity gradients for Ra $= 2 \cdot 10^7$. The dashed line stands for the DNS results, the solid line for our LES7RK results and the dotted one for the LES7PISO results.

vertical profile of the DNS. The maximum value is a little bit lower and closer to the top and bottom plates for both LES7RK and LES7PISO cases. However, we can observe that in the LES7RK case the profile of $(1/Re)(\delta u_t/\delta n)$ is not symmetrical in the upper half of the domain compared to the lower.

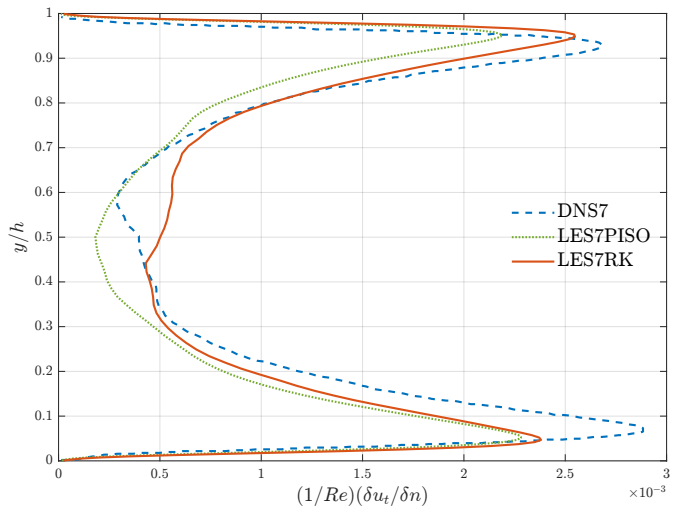


Figure 1.15 Vertical profiles at the sidewall of the mean (the profile is averaged in time and the azimuthal direction) wall-normalized velocity gradients for Ra $= 2 \cdot 10^7$. The dashed line stands for the DNS results, the solid line for our LES7RK results and the dotted one for the LES7PISO results.

The mean temperature profiles clearly show that the temperature remains approximately constant within the bulk region while the temperature gradients are effective only within the thermal boundary layer. As was previously outlined, the temperature gradient is used for the calculation of Nu_{plate}. Consequently, the lower the slope of the curve, the higher the temperature gradient and the Nu_{plate} will be. A close-up view of the $\langle\theta\rangle$ profiles near the upper plate is shown in figure 1.16. The mean temperature for LES7RK and LES7PISO are compared with the DNS7 results of [Verzicco and Camussi,

2003]. The data shows there is a good agreement for LES7RK with the higher error of 5% at $y/h = 0.99$. In the LES7PISO case, the results are less accurate.

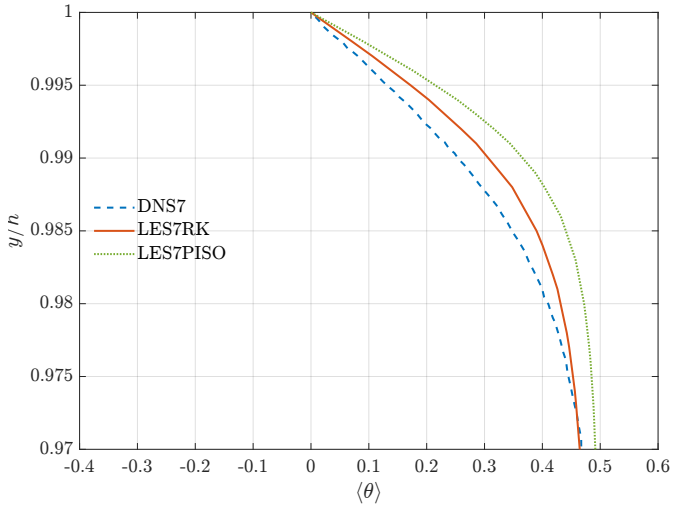


Figure 1.16 Mean vertical profiles of the temperature $\langle\theta\rangle$ at the axis of the cell plotted in the upper half of the domain. The dashed line stands for the DNS results, the solid line for our LES7RK results and the dotted one for the LES7PISO results.

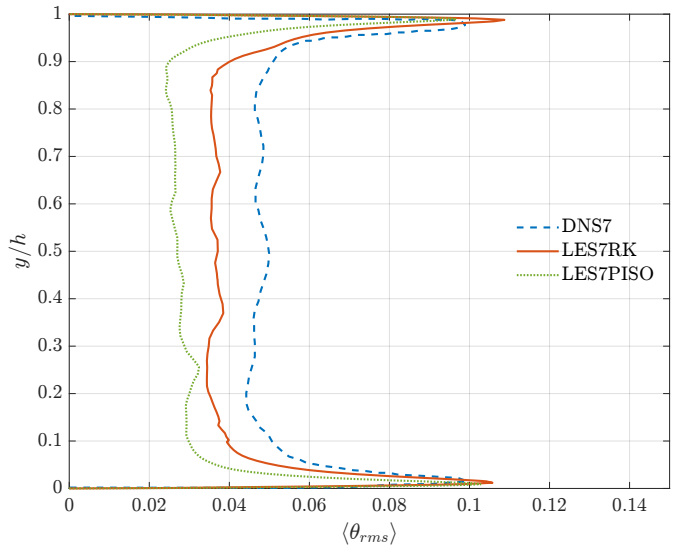


Figure 1.17 *rms* vertical profile of temperature $\langle\theta_{rms}\rangle$ at the axis of the cell. The dashed line stands for the DNS results, the solid line for our LES7RK results and the dotted one for the LES7PISO results.

The dynamics of the mean flow reflect on the dynamics of the boundary layers just as the descending plume at the center of the cell shrinks the boundary layer. This induces a boundary layer that sets monotonically while spreading over the plates across the cross-section aligned with the mean current. In the following figures, the *rms* vertical profiles of vertical velocity and temperature are reported. In particular, in figure (1.17) the *rms* temperature profile at the axis of the cell is presented.

The vertical profile of the temperature fluctuations have peaks in the regions close to the wall and an almost constant value in the bulk of the flow. Both LES cases underestimate the value in the central region but give nearly equal peaks near the plates. In the center of the domain, the differences between LES7RK and LES7PISO

are more pronounced, with the former always more accurate than the latter. To check the estimation of the thermal boundary layer thickness obtained with equation (1.9) we can compare it with the distance of the peak *rms* from the wall, as shown in figure 1.18.

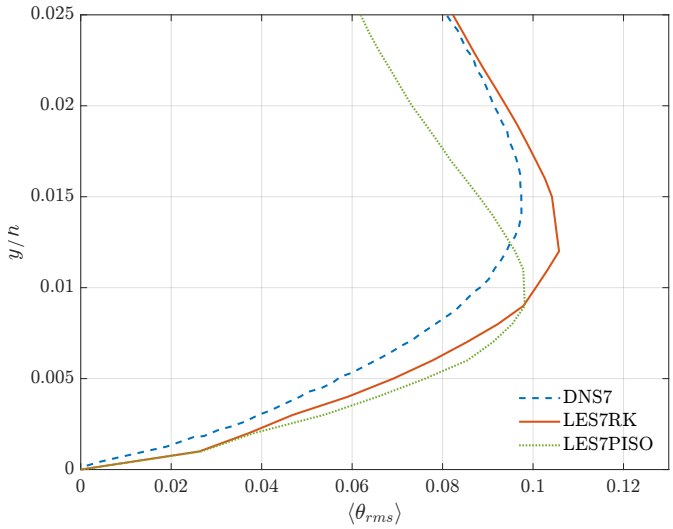


Figure 1.18 *rms* vertical profile of temperature $\langle\theta_{rms}\rangle$ at the axis of the cell near the lower plate. The dashed line stands for the DNS results, the solid line for our LES7RK results and the dotted one for the LES7PISO results.

The profile obtained near the wall using LES7RK is comparable with the DNS trendline but overestimates the peak value. On the other hand, with the PISO results the same peak value is obtained but it occurs at a lower domain height. As we move away from the wall, the DNS results are largely underestimated. The same general considerations can be made for the *rms* velocity profile $\langle u_{rms}\rangle$. It has peaks near the walls and a lower, but not so constant as $\langle\theta_{rms}\rangle$, value in the bulk region. Given that the finding of the first- and second-order statistics already discussed for Ra$= 2 \cdot 10^7$, we can conclude that with RK4 we can obtain results that greatly reproduce those of [Verzicco and Camussi, 2003]. For this reason, the analysis of mean fields and their fluctuations for higher Ra numbers will be carried out with the RK4 algorithm for the discretization of the governing Navier-Stokes equations.

The simulations were carried out using 14 processors and, as expected, the computational cost for the two solvers is different. With the *buoyantBoussinesqPisoFoam* solver the time required for each time step, the latter fixed at 0.001 s, is 1.45 seconds. It means that to perform a complete case with Ra$= 2 \cdot 10^7$ for a total of 595 seconds, with the PISO solver the total simulation time is nearly 240 hours. However, using the *buoyantBoussinesqRungeKuttaFoam* solver with the same time step and the same total time of simulation, the time required for each step is 1.83 seconds. In this case, we need 302 hours to get a complete simulation. Multiplying by the number of processors used, for the PISO solver we have 3360 total hours while for the RK4 solver are about 4228 hours. The above data and data from all other simulations that we performed are summarized in table 1.7.

Table 1.7 Computational costs of simulations for Ra$= 2 \cdot 10^7$ and for Ra$= 2 \cdot 10^{10}$. With cpu_{proc} we indicate the number of processors used, Δt_{sim} is the time step of the simulations, T_{tot}^{sim} is the total time of the simulations, Δt_{iter} is the time required for each time step, T_{tot}^{real} is the time required to get the complete simulation and *cpu time total* is the product of T_{tot}^{real} and the number of processors used.

	cpu_{proc}	$\Delta t_{sim}[s]$	$T_{tot}^{sim}[s]$	$\Delta t_{iter}[s]$	$T_{tot}^{real}[h]$	*cpu time total* $[h]$
LES7PISO	14	0.001	595	1.45	240	3360
LES7PISO1	14	0.001	595	1.45	240	3360
LES7RK	14	0.001	595	1.83	302	4228
LES7RK1	14	0.001	595	1.83	302	4228
LES10RK	16	0.001	900	2.00	500	8000
LES10RK1	16	0.001	900	2.00	500	8000

1.4.2 Wave loads over fixed rectangular pontoon

OpenFOAM's performance will be assessed on a test case that simulates water waves and the interaction between waves and fixed structures. The package *OlaFlow* [Higuera, 2017] included in OpenFOAM v2206 is utilised. In brief, *OlaFlow* solves the incompressible Navier-Stokes equations (hereafter rewritten for sake of clarity):

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{1.12}$$

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_j} + \nu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + g_i \, . \tag{1.13}$$

The frame of reference has x (or x_1) set along the direction of wave propagation, y (or x_2) in the cross-stream direction, and x_3 (or z) in the vertical upward direction. The gravitational acceleration vector is $g_i \equiv (0,0,-g)$. The set of equations is solved in the liquid phase and the air phase, and the VOF method is used to detect the interface, using a parameter α governed by the equation:

$$\frac{\partial \alpha}{\partial t} + \frac{1}{\theta}\frac{\partial \langle u_i \rangle \alpha}{\partial x_i} + \frac{1}{\theta}\frac{\partial \langle u_{c_i} \rangle \alpha(1-\alpha)}{\partial x_i} = 0 \, . \tag{1.14}$$

The parameter α represents the fraction of liquid contained within a cell. $\alpha = 1$ if the cell is full of liquid and, conversely, $\alpha = 0$ for gas cells. u_{c_i} is the difference between fluid velocity in the liquid phase and fluid velocity in the gas phase. The equations are solved using the PIMPLE algorithm. As boundary conditions, a wave generator is placed on the left side of the domain ($x = 0$), where the values of velocity and VOF function (field *alpha*) are set according to wave theories. On the bottom wall no-slip velocity is considered, at the right boundary of the domain wave absorption is used. In order to cancel out the reflected waves, the boundary must generate a velocity equal to the incident one but in the opposite direction. The later walls are *empty*, a condition used in OpenFOAM 2D simulations. The top boundary is set to *pressureInletOutletVelocity* for the velocity, which allows exit from the domain, and *totalPressure* for the pressure, which allows uniform pressure on the patch. At the interface of two different fluids the equation (1.14) keeps the surface compressed. The solver allows the generation of large amplitude waves, which, depending on the period, the height and the depth of the liquid phase, are described by various theories (Cnoidal, StokesI, StokesII, etc.).

The mesh, as reported by Zhang et al. [Zhang et al., 2018], must be sufficiently refined in the region of the free surface, to allow a correct representation of *alpha.water*. The mesh must be refined according to the height of the wave and the length of the wave. For the vertical dimension of the mesh, the cells must be at most 1/60 of the wave height, while for the horizontal dimension, they must be 1/100 of the wave length. As a first step, the accuracy of the wave generator module (*OlaFlow*) was verified by generating a second-order Stokes wave with a height of 0.2 m and a period of 3 s in a domain with a depth of 4 m.

Table 1.8 Wave characteristics and mesh parameters for the initial test cases.

Case	Wave characteristics			Cell dimension		Mesh	
	H_i [mm]	L_i [m]	z [mm]	x [m]	z [m]	n cells	n points
1	0.75	19.7	12.5	0.1970	0.0125	717192	1438966
2	0.5	14.66	8.33	0.1466	8.3×10^{-3}	784192	1573166
3	0.4	12.26	6.67	0.1225	6.67×10^{-3}	877992	1761054
4	0.3	9.63	5	0.0963	5×10^{-3}	923354	1854564
5	0.2	6.69	3.33	0.0669	3.33×10^{-3}	945564	1934234

The time step was adjustable in reported simulations in order to maintain the Courant number $Co < 0.5$. The maximum allowed time step was 0.025 s. The wave was recorded in post-processing by means of a *sample* to identify the time record of the free surface; successively, data analysis was carried out using a Matlab script. The verification, carried out by overlapping an analytical curve of the second-order Stokes wave, yielded good results, with approximately 0.4% deviation at some points. Figure 1.19 depicts the results. The verification process indicated the need to analyse the fields at least every 0.02 s to avoid missing wave peaks.

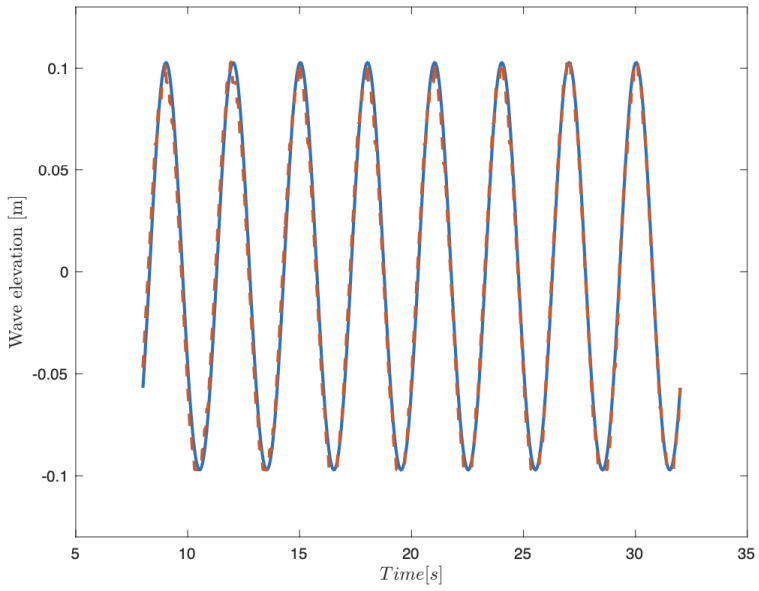


Figure 1.19 Comparison of the analytical and the generated wave. The solid blue line represents the analytical wave and the dashed orange the recorded wave.

Table 1.9 Computational times for considered test cases. Δt_{last} is the last time step of the simulation as the simulations are time step adjustable. T_{tot}^{sim} is the simulation time. Δt_{iter} is the time required for each time step. T_{tot}^{real} is total computational time. T_{total} is cummulative CPU time.

Case	n CPU	Δt_{last} [s]	T_{tot}^{sim} [s]	Δt_{iter} [s]	T_{tot}^{real} [h]	T_{total} [h]
1	8	0.0024	40	1	3.6	28.8
2	8	0.0053	40	1	4.2	33.6
3	8	0.0017	40	1	4.72	37.76
4	8	0.0062	40	1	5.28	42.24
5	8	0.0034	40	1	6.87	54.96

Next, we considered the case of a box-shaped wave breaker whose geometry and position is given in figure 1.20. We considered 5 different waves (see table 1.10).

Table 1.10 Waves generated in the simulations.

Case	H [m]	T [s]
WS 01	0.75	3.76
WS 02	0.5	3.13
WS 03	0.4	2.83
WS 04	0.3	2.49
WS 05	0.2	2.07

Initially, the simulations were carried out in laminar mode. Successively they were switched to the RANS mode with a $\kappa - \epsilon$ turbulence model to analyse the energy dissipation during the interaction between water and structure. The domain was 60 m long and 7 m high with the free surface set at 2 m. The wave generation inlet was set at the left side and the wave absorption outlet at the right side. The top of the boundary was set as the atmosphere, while the bottom is a wall as is the structure of the breakwaters.

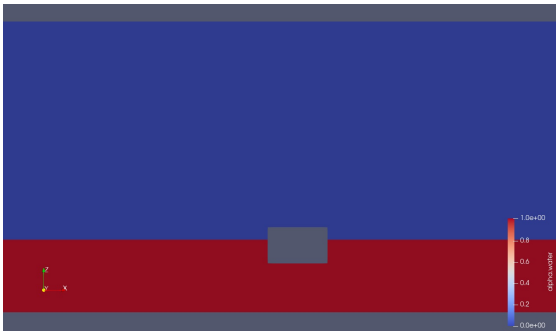


Figure 1.20 Rectangular fixed breakwater.

The performance of floating breakwaters can be studied with the transmission coefficient, which is the ratio between the height of the transmitted wave and the height of the incident wave. To analyze the transmitted wave, data from the free surface were recorded at $L_{fs} = 10$ m downstream of the wave breaker for all five simulations with different waves. The transmission coefficients were compared with [Zhang et al., 2018] as shown in figure 1.21, wherein the x-axis there is the ratio between the length of the

breakwater and the length of the wave BL and in the y-axis the transmission coefficient.

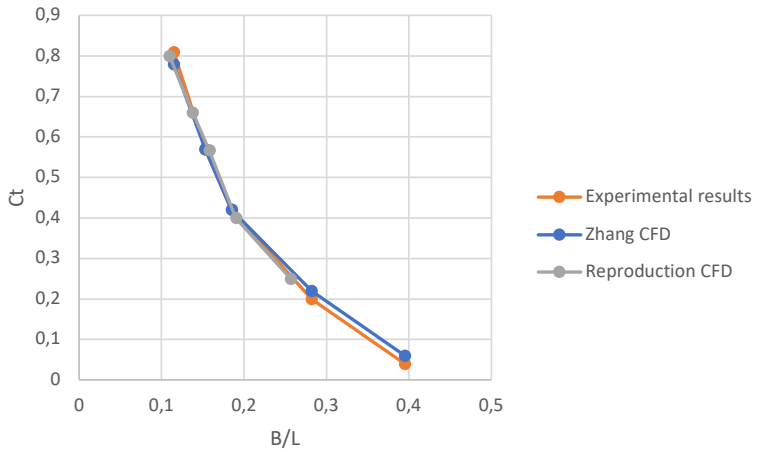


Figure 1.21 Comparison of the transimission coefficients between the CFD and the [Zhang et al., 2018] results.

Additionally, the transmission coefficients for a fixed π shaped breakwater based on the data for wave generation in table 1.10 were analysed. This simulation is done to confirm that by increasing the submergence of the structure, the transmission coefficient changes. The shape of the breakwater is shown in figure 1.22. A wave breaking over the structure is shown in figure 1.23.

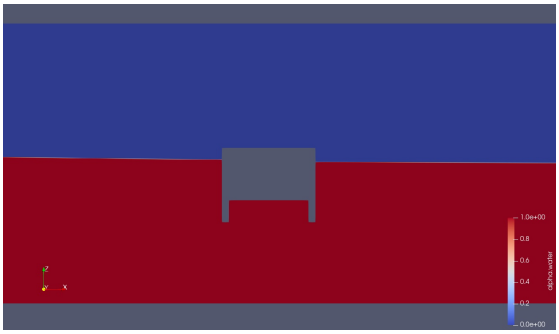


Figure 1.22 π shape breakwaters.

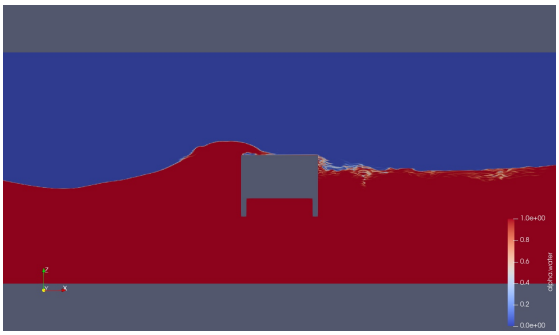


Figure 1.23 Breakwaters with wave load.

The domain in conducted simulations was 70 m long and 9 m high with the free surface set at 4.5 m. Similarly, the wave generation inlet is set at the left side and

the wave absorption outlet at the right side. The top of the boundary is set as the atmosphere, while the bottom is a wall as is the structure of the breakwaters. The problem is two dimensional. The results indicated an improvement in the transmission coefficient and were compared to the empirical formulas from [Macagno, 1953] and [Ruol et al., 2013] that are designed respectively for box-shaped floating breakwaters and π floating breakwaters. The coefficients calculated from the simulation are higher than those derived from the analytical formulae. That's because the structure is fixed in space and not floating on the water, hence energy dissipation is higher.

Subsequently, a comparison between a cubic breakwater and the π-shaped breakwater. The results showed that due to the irregular shape of the π-shaped structure, more energy was dissipated as the wave passed through. Figure 1.24 shows the differences in the calculated transmission coefficients.

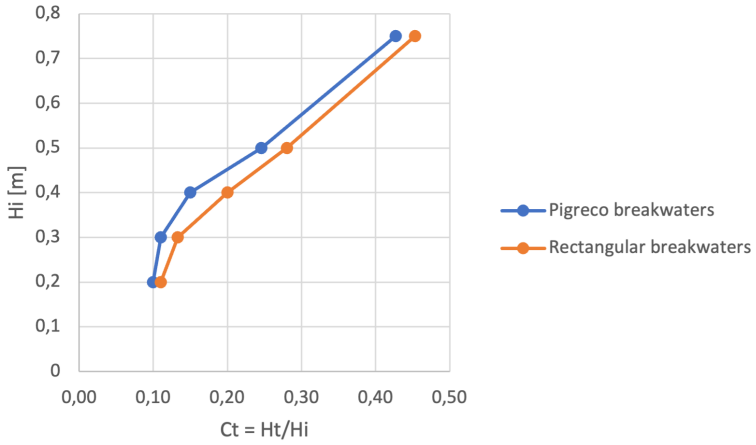


Figure 1.24 Transmission coefficient C_t as a function of incident wave height H_i.

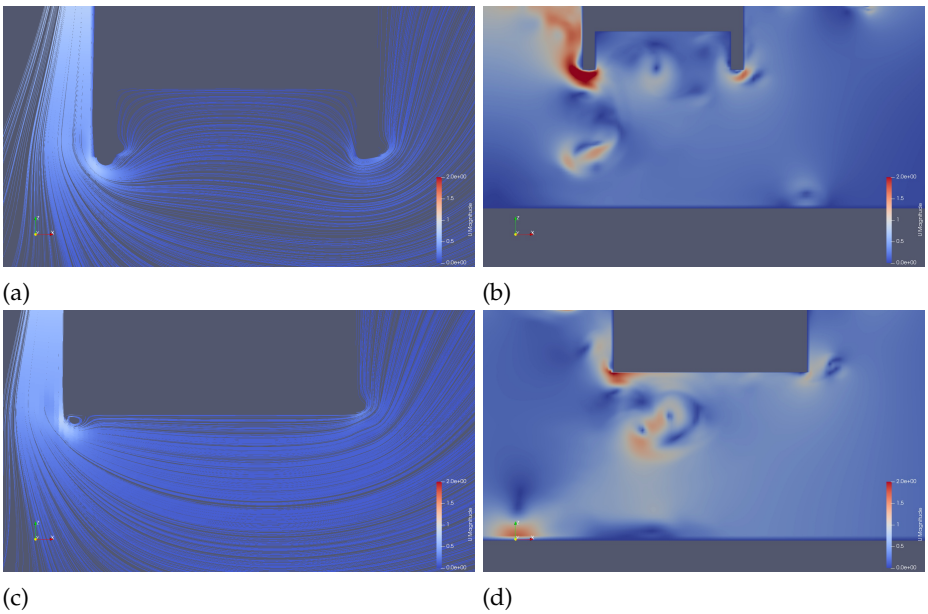


Figure 1.25 Streamlines (a) and velocity magnitude (b) under the π-shaped structure. Streamlines (c) and velocity magnitude (d) under the cubic-shaped structure.

For the same incident wave, the transmitted waves for the π-shaped breakwater are

lower than for the cubic wave breaker. In figure 1.25 streamlines and velocity contours are shown. A large vortex can be observed under the π-shaped structure.

Finally, we will compare the performance of two identical breakwaters geometries. The only difference is in the fact that the new simulation allows the floating breakwater to move vertically (z-direction) while maintaining the waterline similar to the case of the fixed breakwater in space. As far as the floating breaker is concerned, first, the damping of the structure was calculated. To do this, the cube was placed in an unbalanced position, namely, the water surface was set 10 cm above the body's hydro-static balance. The phenomenon is shown in figure 1.26. The wave-structure iterations were then analysed, comparing the displacements recorded on the structure and the analytically calculated displacements. In addition, the oscillation frequencies of the wave-structure system were recorded and subsequently compared with the analytical formulae. The results are shown in figure 1.27.

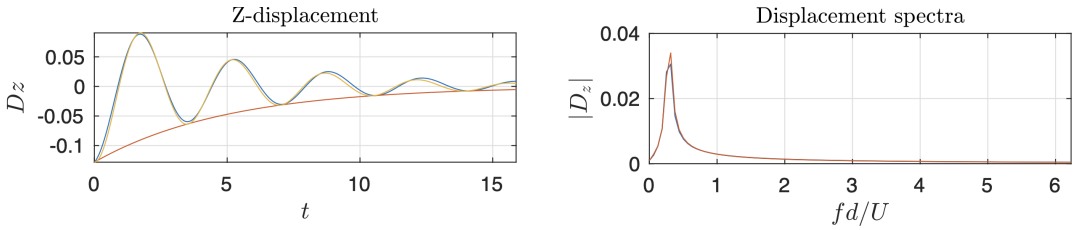


Figure 1.26 Logarithmic trend of the damping simulation.

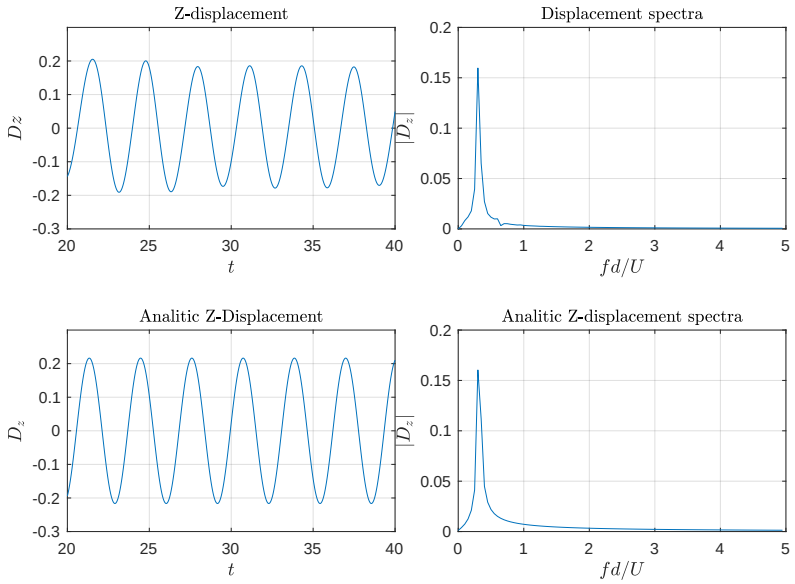


Figure 1.27 Displacement of the breakwater during wave-structure interaction.

The wave transmission coefficient was also calculated for the floating wave breaker, recording the free surface 10 m after the structure. The results are shown in Figure 1.28.

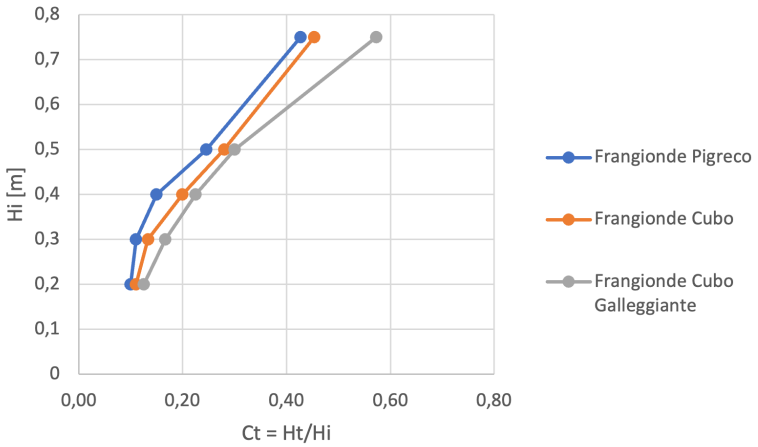
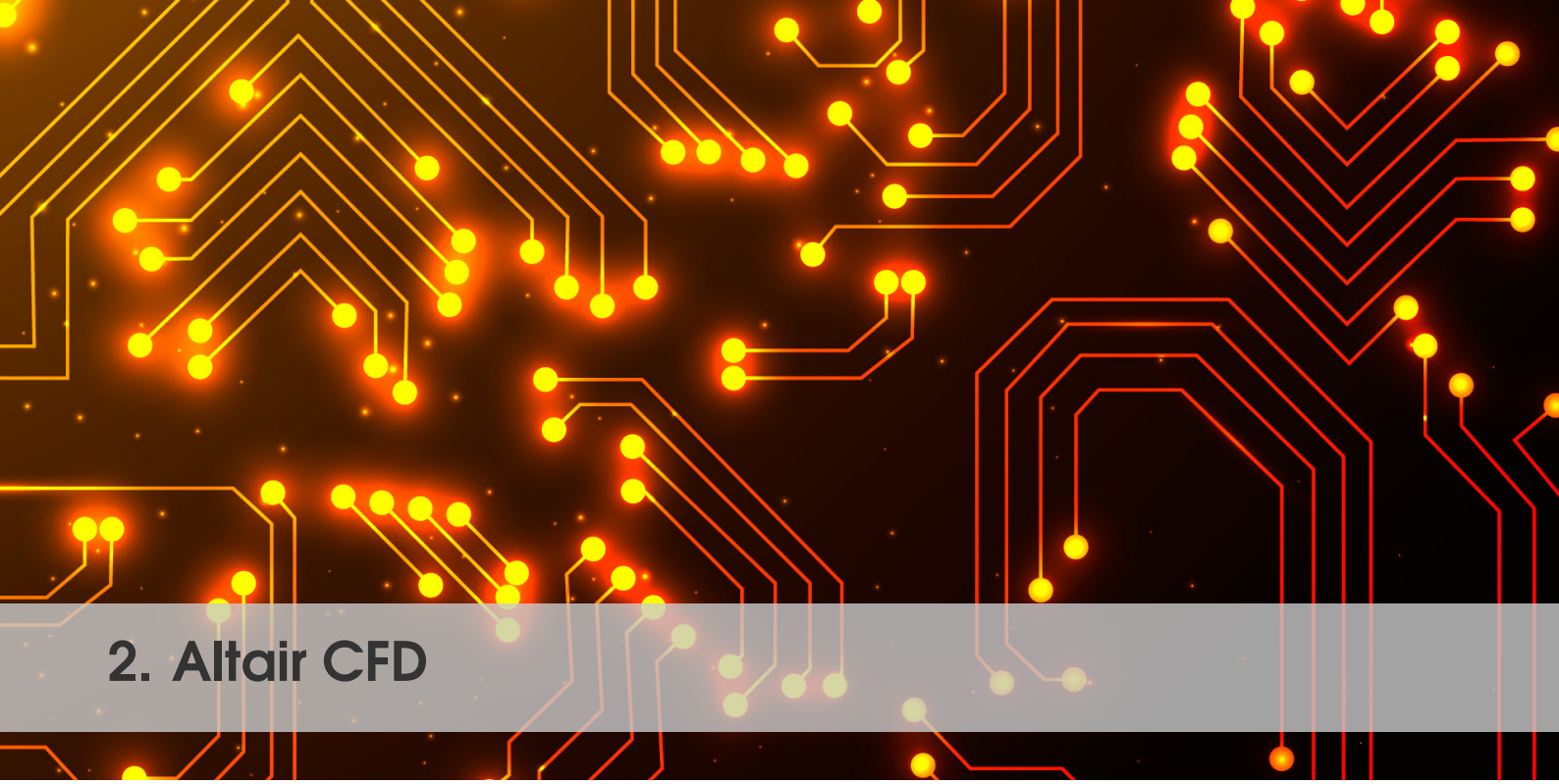


Figure 1.28 Comparison of the transmission coefficients between the floating breakwater and the two fixed structures with the shape of π and cube.



2. Altair CFD

Introduction
Numerical model
Performance comparison

2.1 Introduction

University of Rijeka

The improvement in computational performance, particularly related to graphic processing units, has led to the increased popularity of numerical methods such as the Lattice Boltzmann Method (LBM) [Kruger et al., 2017]. In this chapter, we will compare the conventional macroscopic approach based on FEM with the LBM mesoscopic approach, which considers fluid flow at a smaller scale. We will assess the FEM approach using Altair AcuSolve software and investigate the LBM approach using Altair's UltraFluidX GPGPU-based solver. LBM, in comparison to FEM which uses elements, is based on lattices, where the statistical Boltzmann equation is analyzed through the collision and propagation of particles between lattices.

The FEM-based solver was deployed on the BURA supercomputer cluster, while LBM-based solver was assessed on Nvidia RTX A6000 and Quadro M6000 powered systems. For comparison purposes, we use the external fluid flow problem around the NACA0012 airfoil. The pressure coefficient around the NACA0012 airfoil for a medium Reynolds number of around 190,000 is used to validate these approaches (figure 2.1).

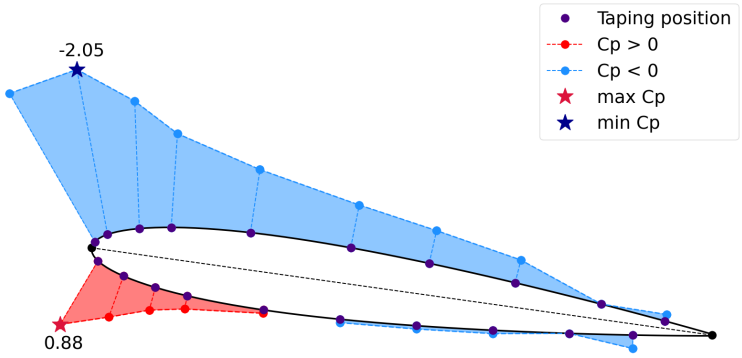


Figure 2.1 Pressure distribution over NACA0012 airfoil for 8° angle of attack for wind tunnel experiment. Minimal and maximal values are highlighted with dark red and dark blue star.

Static pressures were obtained from the twenty tapping positions on the cross-section of the NACA0012 airfoil used in the wind tunnel experiment. Detailed analysis of LBM regarding wind tunnel experiment results is elaborated in a research paper by Rak et al. [2023]. These pressure values were then used to calculate the coefficient pressure around the airfoil, which will serve as the basis for comparing the LBM and FEM approaches.

2.2 Numerical model

The reported case considers the NACA0012 airfoil with an 8° angle of attack (AoA) in a test section with a width of 300mm, height of 300mm, and length of 600mm. The geometry is defined based on the conducted experiment. Both the LBM and FEM approaches use LES with a sub-grid scale Smagorinsky model and share the same inputs, including a 20 m/s inlet velocity, the same air properties, and the same run time simulation. The last 10% of the obtained pressure coefficient values for both cases are time-averaged and compared. Once the coefficient pressure values have been

compared and deemed satisfactory, we will analyze the hardware usage in more detail to gain insights into the performance of the FEM and LBM approaches on the different architectures.

With regard to preprocessing, the LBM approach is more intuitive and easier to deploy due to its use of a lattice layout. The domain can be defined with different amounts of lattices using boundary boxes and body offsets. To determine the optimal lattice distribution, several meshes with different refinement levels were analyzed. The finest mesh, containing around 17 million voxels, was selected based on its correlation with the experimental results. The Root-Mean-Square-Error (RMSE) was found to be lower than 0.15 for this mesh.

In contrast to LBM, the conventional CFD approach based on FEM required a quality mesh. In this instance, preprocessing, i.e. building a numerical grid, can be a time-consuming process. We initially performed a RANS simulation with the k-ω SST model. Subsequently, a fine mesh with around 8.5 million elements was chosen to ensure mesh relevance and minimize numerical errors. The inlet, walls, and outlet boundary conditions were the same for both LBM and FEM simulations. By analyzing the results and visualizing the flow with contour plots (figures 2.2 and 2.3), we can draw conclusions about the quality of the simulations. Following the validation process, we can thus proceed with hardware and performance comparisons.

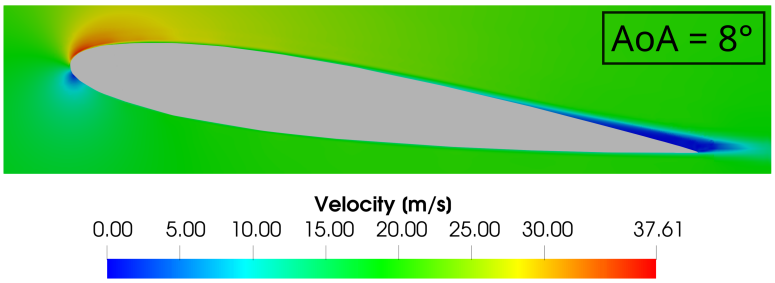


Figure 2.2 Contour plot of the velocity field for the NACA0012 airfoil.

Figure 2.2 shows a zoomed-in velocity contour plot, which reveals a velocity drop near the trailing edge of the airfoil (back part of the airfoil), indicating the presence of the separation. The stagnation point at the leading edge (front part of the airfoil), where the velocity is zero and the pressure is maximal, can be identified easily. Figure 2.3 displays a zoomed-in pressure plot, which illustrates the pressure difference between the lower and upper surface of the airfoil, generating the lift force. Overall, by examining plots we can conclude that the pressure and velocity fields around the airfoil appear reasonable.

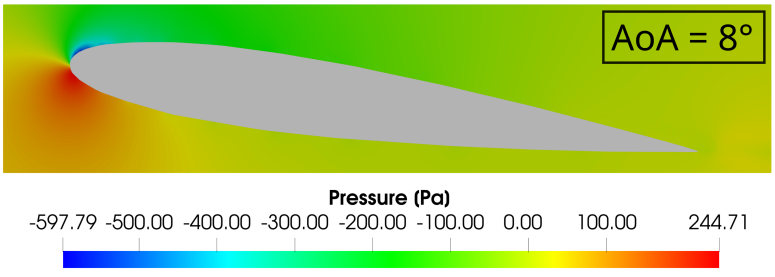


Figure 2.3 Contour plot of the pressure field for the NACA0012 airfoil.

2.3 Performance comparison

The LBM simulation was performed using GPUs. Initially, we used an Nvidia Quadro M6000 device with 12 GB memory, which took a little less than a day to produce the results. Significant improvements were observed by switching to the Nvidia RTX A6000 device with 48 GB memory. In fact, to solve the identical problem, the newer architecture reduced the computational time by threefold compared to the Nvidia Quadro M6000 processor. Furthermore, we achieved even better results by utilizing two Nvidia RTX A6000 devices connected via the NVLink bridge. This configuration not only added more resources but also enabled faster data transfer and code control between GPUs through the NVLink bridge, leading to further reductions in computational time.

It is interesting to investigate the financial aspect of the hardware used to conduct analyses. In this particular case, we can refer to MSRP prices as well as consider current prices. According to the current GPU market, the Nvidia Quadro M6000 costs around $500, while the Nvidia RTX A6000 costs around $4000. MSRP prices for noted devices were $5000 and $4649 respectively. Evidently, faster computational times require more investment in hardware architecture, however, if assessed by the MSRP at the time of introduction, performance has over the years increased with the price remaining approximately the same, or even slightly lower. The correlation between prices and performance is illustrated in Figure 2.4.

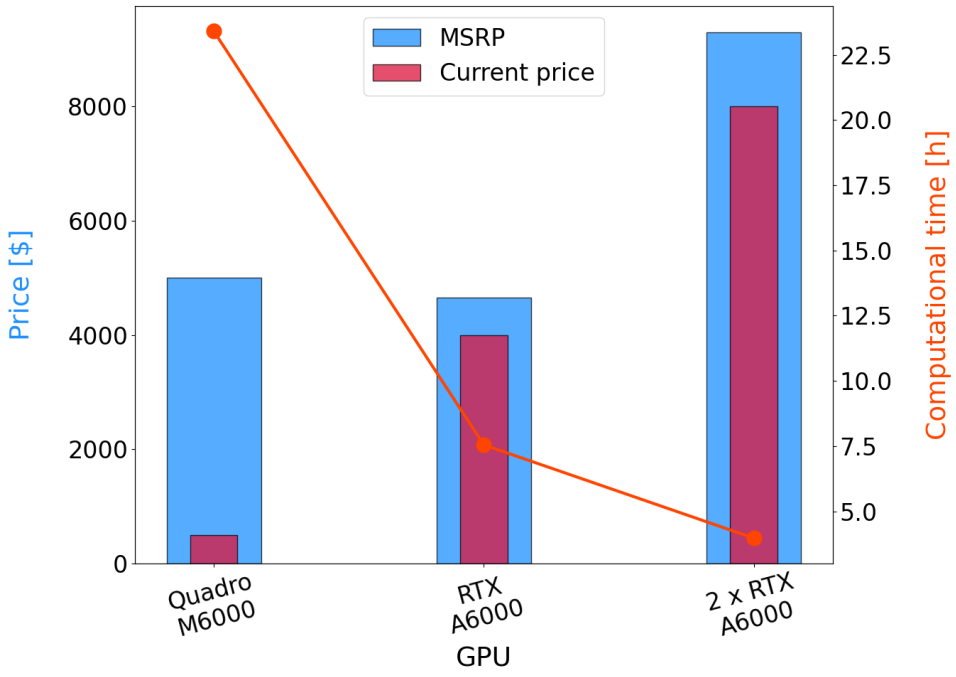


Figure 2.4 Comparison of GPUs based on their market value and computational time. The orange line represents the computational time, while bars show the price of GPUs in US dollars.

Generational improvement in GPU performance in the last five years (the Quadro M6000 was released in 2015 and the RTX A6000 in 2020) is noticeable. If we are to consider current market prices and performance, we are paying eight times more for roughly four times the performance. However, if we are to consider MSRP, this performance can actually be obtained at a slightly lower price. It is also worth noting that buying dated hardware such as M6000 is not a worthwhile investment and

alternatives from Nvidia or AMD should be considered. Additionally, the use of two RTX A6000 GPUs coupled with an NVLink bridge can result in an almost twofold reduction in simulation elapsed time.

As noted, the FEM-based solver was assessed on the BURA supercomputer, which is equipped with Intel Xeon E5-2690 v3 processors. Each processor has a 30 MB cache and 12 physical cores. There are two processors in a node and each node has 64 GB DDR4 2133MHz memory. Nodes are connected to shared petabyte storage with the Lustre file system. For conducted runs, up to 20 physical cores per node were allocated and hyper-threading was disabled. Set simulation time is one-tenth of the total time used for LBM simulations hence it is reasonable to assume that a realistic computational time would be approximately ten times longer. This choice was done to compute results in a reasonable time frame.

In terms of cost, each processor currently has a price of approximately $200. The MSRP for a single E5-2690 v3 was $2090. This totals to $400 and $4180 per pair of CPUs per node, respectively. The relationship between price and performance is illustrated in Figure 2.5.

Figure 2.5 Price and computational time as a function of the number of processors. The light orange line represents the computational time, while the sky blue bars show the price of CPUs in US dollars.

The price bar is continuously increasing with the increase in the number of CPUs. However, the computational time did not decrease linearly. This can be attributed to multiple reasons, including inter and intra-nodal communication, storage latency as well as general solver-related limitations. The most significant decrease in performance is observed when using more than ten nodes. Therefore, for this problem, adding more than twelve nodes would only marginally improve the speedup while significantly increasing the budget.

We have additionally compared CPU and GPU effectiveness in terms of hardware cost and computational time. This data is given in figure 2.6. It's worth noting that the CPU computational time for each simulation has been multiplied by ten to approximate the final elapsed time for the full simulation. When comparing the two approaches, we can conclude that with 24 CPUs, the simulation is completed in approximately one week, while with the Quadro M6000 GPU, it can be completed in just one day. From a

financial effectiveness standpoint, the FEM-based approach requires CPU investment of roughly $4180 according to MSRP, while the Quadro M6000 GPU costs $5000. If we take into account current pricing, the gap is even narrower. To summarize, the LBM-based GPU approach yields results approximately an order of a magnitude faster while being cheaper. This is without taking into account current hardware options (i.e. RTX A6000), though to conduct a fair comparison, equivalent current-generation CPUs should be considered.

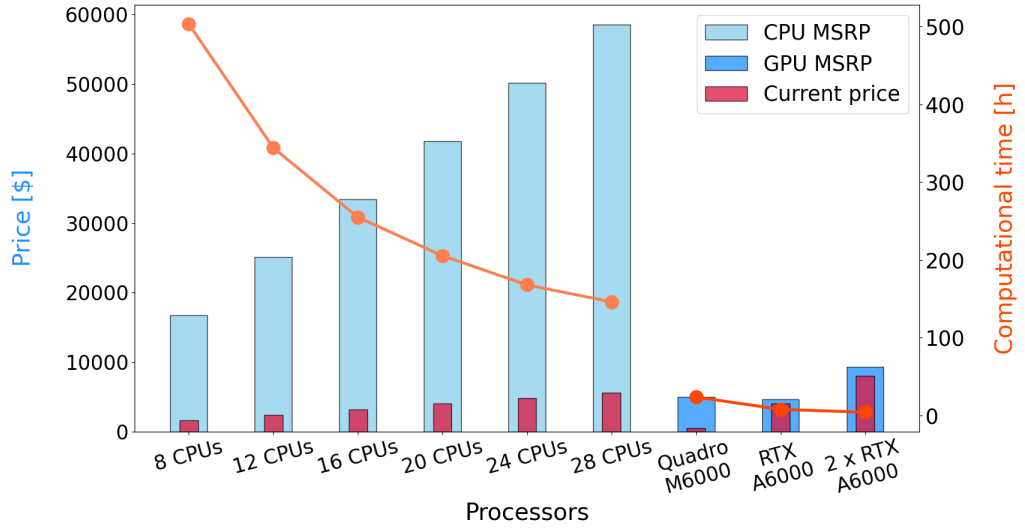
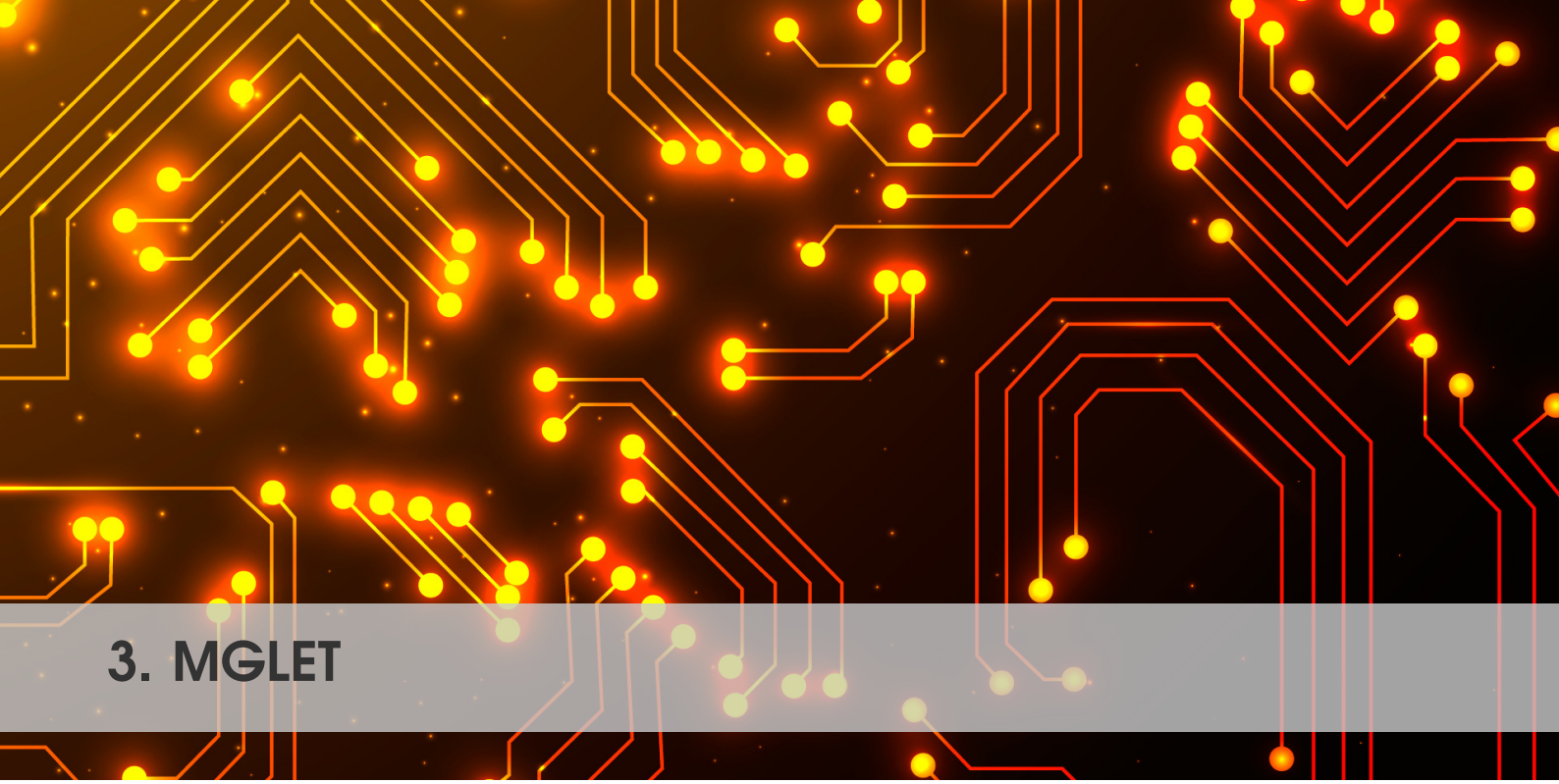


Figure 2.6 Comparative assessment of market value and computational time for considered CPUs and GPUs.

Despite the significant advantage of using LBM on GPUs, there are still scenarios where FEM-based and FVM-based solvers might be preferred. LBM on GPUs can produce results much more quickly and has a shorter preprocessing time, but it may not always be the best approach for solving different types of problems. On the other hand, conventional methods are more robust and have been thoroughly researched. Nevertheless, in this specific case, LBM has proven to be a superior approach. Likewise, it will be interesting to see how the future growth of GPUs will impact the direction of CFD approaches for solving different types of problems.



3. MGLET

CFD code MGLET
Applications
Performance optimisations

3.1 CFD code MGLET

Technical University of Munich

MGLET is a (soon-to-be) open-source CFD software that is capable of performing *Direct Numerical Simulations* (DNS) and *Large-Eddy Simulations* (LES) of turbulent flows in arbitrary-shaped domains, which can be optionally coupled with transport of multiple scalar quantities. The code employs a finite-volume method to solve the incompressible Navier-Stokes equations for the primitive variables. Those variables are stored in a Cartesian grid with a staggered arrangement and discretised in space by a second-order central scheme. The time integration is done by an explicit third-order low-storage Runge-Kutta scheme [Williamson, 1980]. The pressure computation is decoupled from the velocity computation by Chorin's projection method [Chorin, 1968]. Accordingly, a Poisson equation is to be solved for the pressure for each Runge-Kutta sub-step. Arbitrarily curved and geometrically complex surfaces are handled by immersed boundary methods [Peller et al., 2006, Peller, 2010]. A conventional domain decomposition is adopted for parallelisation, which is combined with a local grid refinement strategy. The local refinement is achieved by adding grid boxes with finer resolutions in an octree-like, hierarchical and overlapping manner, where the degree of grid refinement is determined by the grid levels [Manhart, 2004]. This leads to two distinct types of grid-wise communications, namely: *intra-level* communication between the boundaries of neighbouring grids, and *inter-level* boundary and volume communication between adjacent grid levels.

The code is written in Fortran, and the communication between different processes is implemented via Message Passing Interface (MPI). An efficient parallel I/O strategy is implemented based on HDF5 [The HDF Group, 1997-2020]. The pressure computation in the multigrid framework is supported by the hierarchical grid arrangement, and is accomplished by a red-black Gauss-Seidel smoother with over-relaxation (SOR) being applied at all fine grid levels. In contrast, a Strongly Implicit Procedure (SIP) solver is employed at the coarsest level [Stone, 1968]. The usage of two different solvers in the multigrid cycle is justified by the fact that the Gauss-Seidel smoother is effective in eliminating high-frequency error components emerging from the refinement process of the multigrid algorithm, whilst the SIP solver is able to efficiently remove the broad range of frequencies that is present only in the residual at the coarsest level.

3.2 Applications

The code has been developed, maintained and improved by a community of developers and users belonging to several research groups. The group of Prof. H. Andersson and B. Pettersen (NTNU Trondheim, Norway) used the code is primarily for flows around bluff bodies. The group at DLR in Oberpfaffenhofen (Dr. Frak Holzäpfel) use the code for aircraft wake vortices. At the KIT Institute of Meteorology and Climate Research Atmospheric Environmental Research (IMK-IFU, Campus Alpin, Garmisch-Partenkirchen) MGLET is used to assess the sensitivity of ultrasonic flow measurement devices to flow oscillations. The company KMT uses a commercialised version to assess aeroacoustic noise in car passenger compartments. Some recent examples comprise turbulent flow around bluff bodies [Schanderl and Manhart, 2016, Schanderl et al., 2017a,b, Strandenes et al., 2017], aircraft wake vortices [Misaka et al., 2012, Stephan et al., 2014, 2017], unsteady porous media flows [Zhu et al., 2014, Zhu and Manhart,

2016, Sakai and Manhart, 2020, Unglehrt and Manhart, 2022], and fiber suspensions in non-Newtonian fluid media [Moosaie and Manhart, 2013].

More recently, we have been simulating interfacial flows between a layer of a porous medium, representing a sediment bed, and strongly turbulent river stream. We employ a sufficiently large numerical domain and $\mathcal{O}(10^{10})$ grid cells, which are necessary to truthfully capture the very large-scale motions existing in high Reynolds number turbulent flows, whilst resolving all the existing scales of motion (cf. Figure 3.1). The resulting full-resolution data will play a crucial role to improve our understanding of mass and momentum transfer to the sediment bed. Since many processes in the benthic ecosystem are controlled by these parameters, we expect our research to be a new foundation for sustainable river management.

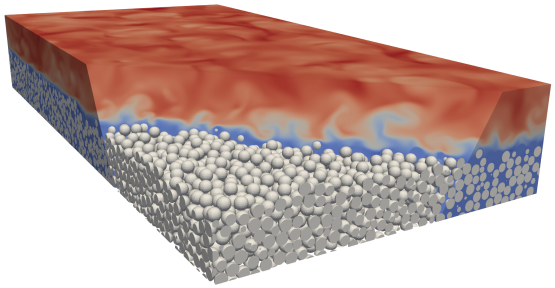
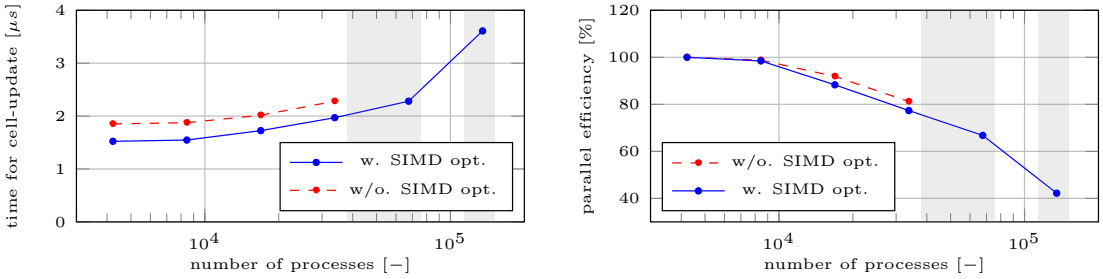


Figure 3.1 Instantaneous velocity field in turbulent flow over a sediment bed.

3.3 Performance optimisations

3.3.1 MPI-level optimisation



(a) Time for cell-update refers to the CPU-time required for one grid cell to advance one step in the time integration scheme.

(b) The parallel efficiency was defined as 100% for the first plotted data point (88 nodes).

Figure 3.2 Weak scaling results plotted over the number of processes on SuperMUC-NG. The problem size is 6.4×10^4 cells/process over a single grid level, i.e. only intra-level boundary communication and SIP pressure solver are relevant. Changes in the background shading indicate a transition of island boundaries.

Over the two performance optimisation projects in 2015 & 2017, MGLET improved its parallel-scaling performance by a factor of ≈ 4 and the I/O performance by a factor of ≈ 25. The details of the MPI-level optimisation as well as the subsequent performance evaluation on 4 different HPC systems can be found in Sakai et al. [2019]. At the time of writing, the code exhibits a satisfactory strong scaling up to a problem size of ≈ 17 billion discrete cells distributed over approximately 32000

parallel processes (cf. partially Sakai et al. [2019]), whilst a sufficient weak scaling was demonstrated up to 135000 parallel processes (cf. Figure 3.2).

3.3.2 SIMD-level optimisation

Consequently, we performed a SIMD optimisation of our two pressure solvers within the third optimisation project in 2019 [Sakai and Manhart, 2021]. This was motivated by the recent trend that modern HPC processors are equipped with ever more powerful yet more energy-efficient internal vectorisation hardware to maintain performance growth while coping with the stagnated nominal frequency, as well as the ever-growing energy consumption for the HPC systems. One important example of such systems for us is SuperMUC-NG at LRZ, which is based on Intel Skylake processors being equipped with 512-bit ultrawide vector registers. SuperMUC-NG achieves the theoretical peak performance at ≈ 27 PFLOPS with the expense of $\approx 1.5 - 2.4$ MW (private communication). By exploiting Skylake's extensive SIMD capability, our optimised code shows up to 20% overall performance improvement even in the range of $\mathcal{O}(10^4)$ MPI processes (cf. Figure 3.2).

Target hardware

The majority of our large-scale simulations are performed on SuperMUC-NG at LRZ, which is based on Intel's Skylake Xeon Platinum 8174 processors. Since the processor is representative of a variety of modern CPU architectures, we defined it as our target hardware for our SIMD optimisation effort. The Skylake processor supports the Fused Multiply-Add (FMA3) as well as the AVX-512 SIMD instruction sets. The processor is equipped with a three-level hierarchical cache, where the L3 cache with a capacity of 33 MB is dynamically shared between all 24 cores on a socket, which constitute a NUMA (Non-Uniform Memory Access) domain. One node consists of two sockets (i.e. 48 physical cores per node), while a fast OmniPath network with 100 Gbit/s is used to connect those nodes. Despite our focus on the above hardware, we also tested the results on different CPUs e.g. on AMD Zen2 architectures.

Mixed precision

As the initial profiling results had shown, the performance of the pressure solver routines of MGLET is largely memory-bound. To obtain accurate high-order statistics for the scientific investigation of turbulent flow cases, the code is usually operated in double precision. Depending on the termination criterion specified by the user, the pressure solver, however, does not benefit from the high-precision floating point representation in most application cases. An option was added allowing users to run the multi-grid pressure solver routines in single precision. Both computation kernels and communication routines benefit from this option. Accordingly, all further improvements use the single precision performance as a baseline.

SOR solver

The multi-grid pressure solver of MGLET employs the Successive Over-Relaxation (SOR) method on the refined grid levels. As such, SOR operates as a smoother to eliminate high-frequency components from the residual. By default, a red-black version of SOR is used which divides the nodes into a red and a black subset. This separation allows a parallel update of one subset because no data dependencies exist internally.

The key concept of our optimization strategy was to increase the density of the parallelly processable data. This approach is successfully demonstrated e.g. by Stürmer [2005] and comprises a data rearrangement as shown in Figure 3.3. Red and black

nodes within a grid box are separated and stored in contiguous sections. Consequently, no capacity of the vector registers is spent on entries that cannot be processed. At the same time, the pressure on the cache hierarchy is reduced since fewer memory pages are requested by the CPU. To decrease the pressure on the cache levels further, the central coefficient of the stencil is computed on the fly. The resulting increased arithmetic intensity accelerates the memory-bound loops.

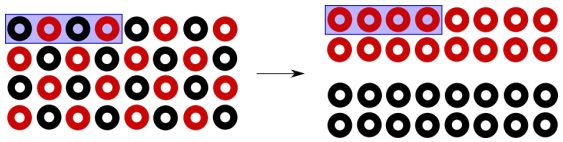


Figure 3.3 Separation of red and black nodes within one grid box. The blue box can be interpreted both as a memory page or the capacity of the vector registers.

The previously described measures increased the performance of the SOR kernel routines. Figure 3.4 shows the wall time consumed for one iteration comprising updates of all red and black nodes. Measurements were made for runs with only 1 active process per socket (2/48) as well as for cases where all physical cores on the node were assigned one process (48/48). Due to shared cache levels, curves for a different occupancy of the node exhibit a different behavior once a certain grid size per process is exceeded. This grid size marks the point where the arrays involved into the iterative solution can no longer be kept in the caches. For the optimized SOR solver, this point is delayed until a value of $64^3 \approx 2.62 \cdot 10^5$ cells per grid are reached.

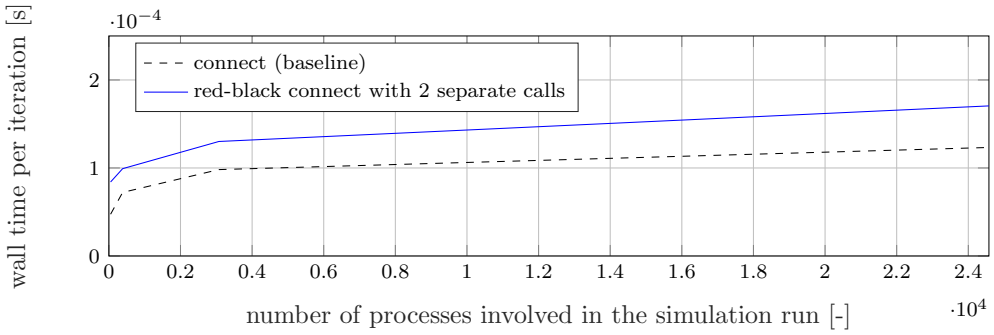


Figure 3.4 Node-level performance of the SOR kernels on Intel Skylake Xeon Platinum 8174. One node carries 2 sockets with 24 cores per socket. Wall times were determined as mean values from 3 independent runs in single precision. Each process operates embarrassingly parallel on one grid (synchronisation only with MPI_Barrier).

SIP solver

The SIP algorithm, also known as Stone's Strongly Implicit Procedure [Stone, 1968], employs an incomplete lower upper decomposition (ILU). Its stability combined with a low number of required iterations are frequently mentioned as the major advantages (e.g.Halada and Lucká [1999] Reeve et al. [2001] Leister and Perić [1994]). Therefore, the multi-grid pressure solver implemented in MGLET uses SIP to solve the Poisson equation on the coarsest grid levels.

Leister and Perić [1994] describe a vectorized version of SIP. This approach uses a wave-front algorithm for three-dimensional applications to resolve the data dependencies during the forward and backward substitution. Dierich et al. [2015] discuss a

combination of SIP and the block Jakobi algorithm. The discretization matrix is split into submatrices which are distributed among different OpenMP threads. Deserno [2003] elaborates on the wave-front algorithm described by Leister and Perić [1994] and refers to it as the *hyperplane* approach. Apart from that, a *hyperline* version is introduced which also serves to avoid data dependencies.

An implementation of the vectorization approaches described in the literature showed a critical deterioration of the performance on several modern vector CPUs. The reason was found in an extremely high rate of cache misses resulting from the irregularly-strided memory access patterns. The underlying idea of our optimization strategy was a reorganization of the data structure. The array entries are rearranged such that unit-strided memory access is possible for the parallelly executable operations within one hyperline. Data re-usage is possible when proceeding from one hyperline to the next. Additionally, the hardware prefetching mechanism is active and helps to reduce the rate of cache misses drastically.

The kernel operations within the SIP solver comprise the computation of the residual as well as the forward and backward substitution. During one iteration cycle, data is retrieved from nine different arrays containing one floating point number for each cell in the grid box. This property of the algorithm puts a considerable load on the memory hierarchy. Figure 3.5 shows the changing performance with increasing amounts of data for two different architectures. On several modern hardware architectures, the L3 cache level is shared between different cores in a NUMA domain. We found that optimal node-level performance for SIP is only reached when the data of all nine arrays fits into the L3 cache. Particularly if all cores on one node are active, this criterion restricts the number of grid cells per core. For a fully occupied Intel Skylake Xeon Platinum 8174 node, an optimal grid size of 40^3 cells was determined. For this case, the wall time for execution of the SIP kernels is halved.

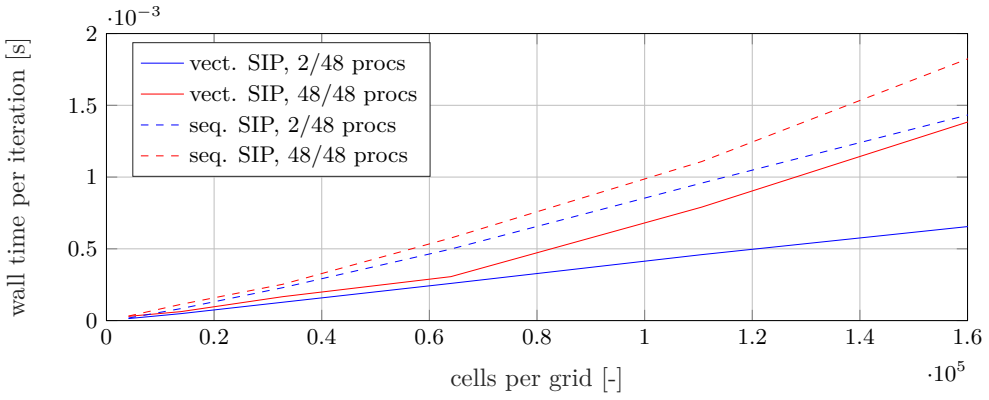


Figure 3.5 Node-level performance of SIP kernels on Intel Skylake Xeon Platinum 8174. One node carries 2 sockets with 24 cores per socket. Wall times were determined as mean values from 3 independent runs in single precision. Each process operates embarrassingly parallel on one grid (synchronisation only with MPI_Barrier).

The MPI communication routines of MGLET had to be adapted to operate efficiently on the reordered data format for the SIP solver. In a first attempt, a custom MPI data type was designed to allow unbuffered communication between grids on one level. Using this strongly irregular data type deteriorated the performance as shown in Figure 3.6. An improvement, however, could be achieved by explicitly copying the data into a contiguous buffer. After the MPI communication with contiguous standard data types, the buffer is unpacked and the data is copied into the hyperline structure.

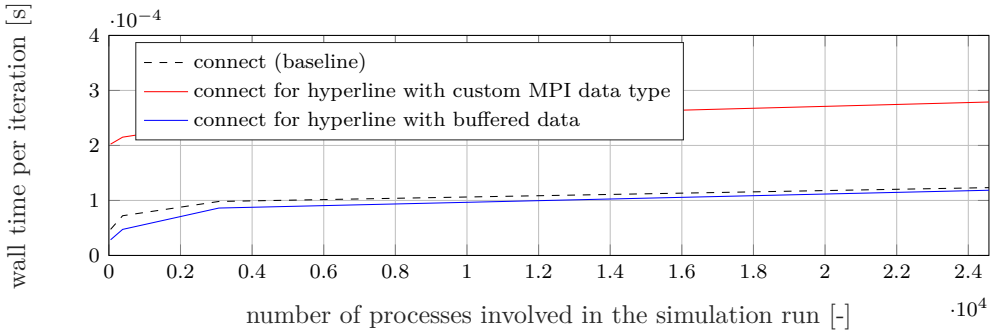


Figure 3.6 Performance of different implementations of `connect()` for the hyperline data arrangement. We used Intel MPI 2019.6.154 on SuperMUC-NG with 48 processes per node. Each process operated on a grid with 64000 cells. Wall times were determined as mean values from 5 independent runs.

3.3.3 GPU optimisation

Our fourth and most recent optimisation effort is dedicated to upgrading the above SIMD-optimised code to be able to run on heterogeneous systems. Though the integrated vector units inside the general-purpose CPUs have significantly improved the performance, or more importantly the performance-energy ratio over the last years, there is a hard limit to the improvement and it may have been reached already. Therefore during the past years, the HPC community has witnessed a persistent trend towards GPU-accelerated heterogeneous systems. As boldly claimed by several experts, the famous Moore's law may have lost its validity, at least for the conventional CPUs. In contrast, the GPU accelerators are equipped with a great degree of thread-level parallelism combined with fast-access memory. Together with their excellent energy efficiency, those characteristics encourage many people to believe that GPUs will play an important role in future exascale computing. For this very reason, even the traditionally-CPU vendors, such as Intel, are now looking into the GPU business. At this point, it is fair to point out that the traditional CFD codes based on the incompressible Navier-Stokes equations, such as MGLET, are generally lagging behind in GPU computing in comparison to the other HPC applications (e.g. compressible Navier-Stokes, molecular dynamics, deep learning solvers). This is due to our specific need to solve a Poisson problem for the pressure which is the most computationally intensive part within the code. The pressure problem features rigid data dependencies of the elliptic nature, which require frequent communications between the subdomains, therefore between different GPUs where the communication bandwidth is critically limited.

4. Tree codes

Introduction
N-body simulations
The tree algorithm
The GADGET4 code
Post-processing tools

4.1 Introduction

Italian National Institute for Astrophysics

Numerical methods play an ever more important role in Astronomy and Astrophysics (A&A). This is true not for theoretical works only, but also for purely observational projects requiring massive use of computational methods. Numerical approaches are viewed as a fundamental complement to analytic reasoning, due to their ability to solve complex systems of equations that are either intractable with analytic approaches or only compliant with highly approximate treatments.

This chapter is meant to provide a general overview of one of the most pertinent techniques widely applied to numerical simulations of galaxy formation and evolution. The discussion is focused on the main numerical concepts rather than on a mathematically detailed exposition of the techniques.

4.2 N-body simulations

In A&A, a N-body simulation is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity. The particles treated by the simulation may or may not correspond to single physical objects. For example, an N-body simulation of a star cluster might have a particle per star, so each particle has some physical significance. On the other hand, describing all the stars in a galaxy as point masses would require order 10^{11} bodies. This may come within reach in a few years, but at present, it is still essentially infeasible on actual HPC platforms. Nevertheless, astronomers describe all the components of galaxies (stars, gas, dark matter) as discrete N-body systems, composed of far fewer particles than there are in reality, trying to find a balance between numerical accuracy and manageable computational requirements.

N-body simulations are simple in principle because they require integrating the $6N$ ordinary differential equations defining the particle motions in Newtonian gravity. In practice, the number N of particles involved in a simulation is usually very large and for each of them, we have to calculate $N-1$ interactions, yielding a computational cost of order $O(N^2)$. Therefore, direct integration of the differential equations is prohibitively computationally expensive for large N. We hence need faster, *approximative* force calculation schemes, which can reduce the computational cost to $O(NlogN)$ or better, at the loss of accuracy. There are different possibilities for this, namely:

- particle-Mesh (PM) algorithm
- Fourier-transform-based solvers of Poisson's equations
- iterative solvers for Poisson's equation (multigrid-methods)
- hierarchical multipole methods (tree-algorithms)
- TreePM methods, hybrid methods that combine both tree and PM algorithms.

In the following, we discuss the tree algorithm that allows us to compute the gravitational forces efficiently.

4.3 The tree algorithm

The Barnes-Hut Algorithm (hereafter BH) describes an effective method for solving N-body problems. It was originally published in 1986 by Josh Barnes and Piet Hut Barnes and Hut [1986]. Instead of directly summing up all forces, it is using an oct-tree-based approximation scheme which reduces the computational complexity of the problem from $O(N^2)$ to $O(N \log N)$. The BH algorithm draws a hierarchical subdivision of space (the volume of the simulation) into cubic cells, each of which is recursively split into eight subcells (hence the name "oct-tree") whenever more than one particle is found in the same cell.

4.3.1 Tree construction

We illustrate what the BH algorithm does in two dimensions, also called quadtree.

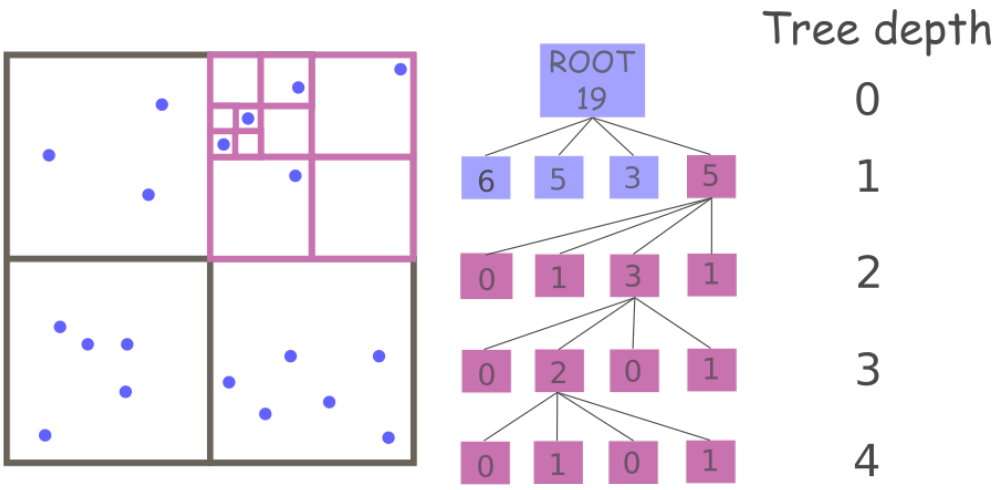


Figure 4.1 Sketch of the tree-construction in two dimensions also called quadtree.

Let's assume that there is a bunch of bodies, e.g. 19 bodies, in a (square) 2D domain, as shown in figure 4.1. This is the full domain and it is represented as the root of the tree. Then the BH algorithm divides the domain evenly in four quadrants (also called nodes), yielding the first level of the tree, and counts how many bodies are contained in each sub-region. According to the prescription provided by the BH algorithm, each sub-region is recursively subdividing until there is at most one body per cell, called a leaf of the tree. Let's focus on the upper right-hand-side quadrant (highlighted in violet) where there are 5 bodies. Four levels are required to reach the bottom of the tree. This process is applied to all quadrants in order to complete the tree of the entire domain.

The pseudo-code to accomplish this task is the following:

Algorithm 4.1 The pseudo-code for the creation of the tree in 2D (also called quadtree). The computational cost of recreating the quadtree depends on the distribution of the particles (tree depth).

```
Function BuildTree
{
  /* free memory if the tree already exists */
  if TreeExists then
    ResetTree

  /* assign a node to each particle     */
  /* starting from the root of the tree */
```

```
  for all particles
  {
    rootNode -> InsertParticle(particle)
  }
}

Function InsertParticle(Particle)
{
   /* if the quadrant is empty just assign the particle to that quadrant */
   if node is empty
   {
      store Particle in node as existingParticle
   }
   /* one particle already in the quadrant */
   else if number of particles in this node == 1
   {
      /* move existing particle to its subquadrant */
      quadrant = GetQuadrant(existingParticle)
      if SubNode(quadrant) does not exist
         create SubNode(quadrant)
      SubNode(quadrant) -> InsertParticle(existingParticle)

      /* assign particle to its subquadrant */
      quadrant = GetQuadrant(Particle)
      if SubNode(quadrant) does not exist
         create SubNode(quadrant)
      SubNode(quadrant) -> InsertParticle(Particle)
   }
   /* open the quadtree and create subquadrants */
   else if number of particles in this node > 1
   {
      quadrant = GetQuadrant(Particle)
      if SubNode(quadrant) does not exist
         create SubNode(quadrant)
      SubNode(quadrant) -> InsertParticle(Particle)
   }

  /* increase the number of the inserted particles */
  Update number of particles
}
```

The cost of adding a particle to the tree is proportional to its distance from the root node (i.e. the depth of the tree). Hence, distributions with many densely packed particles require more operations because the tree must be subdivided often to place particles in their own quadrant.

4.3.2 Computing the mass distribution for each tree node

The building of the hierarchical structure is aimed at force calculations. The next step deals with calculating the mass distribution of the tree, starting from the spatial distribution of the particles contained in the tree. It consists of computing the total mass and the center of mass contained in the child nodes of each tree node. The pseudo-code to calculate the mass distribution of the quadtree is the following:

Algorithm 4.2 The pseudo-code for computing the mass distribution of the quadtree.

```
Function ComputeCenterOfMass
{
    if number of particles in node > 1
```

```
    {
        for all NOT empty child quadrants
        {
            /* get CM and Mass of the current child quadrant */
            Quadrant.ComputeCenterOfMass()

            /* add the mass of the current child quadrant */
            Mass += Quadrant.Mass

            /* update the CM (after the loop over all child */
            /* quadrants it must be divided by the Mass)    */
            CenterOfMass += Quadrant.CenterOfMass * Quadrant.Mass
        }
        CenterOfMass /= Mass
    }
    /* CM of a leaf */
    else if number of particles in node == 1
    {
        CenterOfMass = Particle.Position
        Mass         = Particle.Mass
    }
}
```

The computational cost of this algorithm is of order $O(N \log N)$.

4.3.3 Tree walk and force calculation

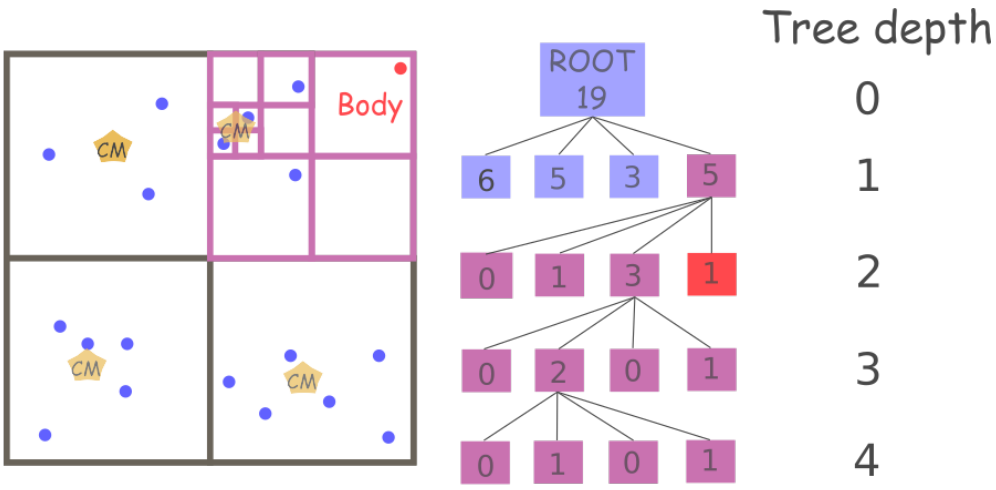


Figure 4.2 Sketch of force evaluation on a single body.

The BH algorithm is faster than the direct summation of all mutual forces in the way it computes the forces exerted on particles. The tree is walked (again) top-down, i.e. starting at the root node. Let l be the size or length of the cell currently being processed, and D the distance from its center of mass (CM) to the target body (the one for which you want to calculate the acting force). If $l/D < \theta$, where θ is an adjustable parameter of the simulation, then approximate the group of bodies in the node by their center of mass, otherwise resolve the node into its subcells and repeat the procedure at the next tree level.

Let's focus on the red body p in figure 4.2. At the first tree level, the size of the 3 cells neighboring the one containing the red body is smaller than the distance to p.

As a result, the force they exert on p is approximated by their center of mass. At the second level of the tree, one cell adjacent p is empty, thus it does not contribute to the force calculation. However, the other cells at the third level contain bodies, two of them must be taken into account individually because are close to p, while the cell of the remaining two is far enough to be treated approximately using the center of mass. Direct force calculation for the p particle would have implied a contribution from 18 bodies, while the BH algorithm reduces the number of force contributions to 6. The accuracy and computational load of the BH algorithm can be tuned by adjusting the θ parameter (direct summation is achieved using $\theta = 0$).

The pseudo-code for the force calculation is then as follows:

Algorithm 4.3 The pseudo-code for computing the gravitational force exerted on a target particle.

```
Function GravitationalForce(targetParticle, theta)
{
    /* total force exerted on targetParticle */
    force = 0

    /* consider all quadrants starting from the top level */
    for all quadrants of rootNode
    {
        /* evaluate the distance between the target    */
        /* particle and the center of mass of the node */
        D = distance from CM of the node to targetParticle

        /* size of current quadrant */
        l = size of the node

        /* criterion is based on the opening angle parameter theta */
        if (l/D < theta)
        {
            /* approximate the group of bodies in the node by their CM */
            force += Gravitational force between targetParticle and node
        }
        else
        {
            /* resolve the node into its subnodes and      */
            /* repeat the procedure at the next tree level */
            for all child quadrants q
            {
                force += q.GravitationalForce(targetParticle)
            }
        }
    }
}
```

Given a simulation with many bodies the processes of i) constructing the tree and ii) for each body walking the tree to calculate the force, must be performed (in principle) at each timestep of the simulation. This is a computationally intensive problem asking to be parallelized.

In the following, we discuss the tree algorithm implementation in the **GADGET4** code and the parallelization strategy.

4.4 The GADGET4 code

GADGET4 is a massively parallel code for N-body/hydrodynamical cosmological simulations. It offers several state-of-the-art simulation algorithms to be applied to a

variety of different types of simulations. An account of all the numerical algorithms used by the code is given in the paper Springel et al. [2021].

The simulation code **GADGET4** (**GA**laxies with **D**ark matter and **G**as int**E**rac**T**) supports collisionless simulations and smoothed particle hydrodynamics on massively parallel computers. The code is written in C++11 standard. Message passing interface (MPI) orchestrates the communications between concurrent execution processes are managed explicitly through MPI, or implicitly through shared-memory accesses on the process on multi-core sockets (the code should be run on parallel platforms MPI-3 compliant). The code is made publicly available under the GNU general public license and obtainable from the public GIT repository `https://gitlab.mpcdf.mpg.de/vrs/gadget4`.

4.4.1 Compilation

Hereafter we describe only what is required in order to compile the code to run a pure N-body simulation (i.e. without hydrodynamics) using the Tree algorithm. **GADGET4** needs the following C++ non-standard library for compilation:

- **MPI**: the Message Passing Interface (version 3.0 or higher). An open-source implementation of such a library is offered by OpenMPI (`https://www.open-mpi.org/`), or MPICH (`https://www.mpich.org/`).
- **gsl**: the GNU scientific library obtainable from `https://www.gnu.org/software/gsl/`.
- **FFTW3** (optional): the Fast Fourier Transform in the West library obtainable from `http://www.fftw.org`. FFTW-3 is only needed if cosmological initial conditions are created. An MPI-capable version of FFTW-3 is not explicitly required.
- **HDF5** (optional): The Hierarchical Data Format (available at `http://hdf.ncsa.uiuc.edu/HDF5`). This library is required only to read or write snapshot files in HDF5 format.

Compilation of **GADGET4** code needs a C++ compiler supporting C++11 standard (for GNU g++ this means version 4.x or later). The code also makes use of GNU-Make and Python as part of its build process.

4.4.2 Building the GADGET4 code

The **GADGET4** code encompasses a bunch of subdirectories and a few further files for the build system in the top-level directory. The most important subdirectory is *src/*, which contains the actual source code files, grouped into subdirectories according to their functionality.

The code is configured by two different files, one containing compile-time options, usually called *Config.sh*, and one listing runtime parameters, usually called *paramfile.txt*. The code uses the GNU make utility for controlling the build process, which is specified by the file *Makefile*.

To compile the **GADGET4** code on a computer system one has to go through the following steps:

- make sure that the required libraries (see Section 4.4.1) are installed and available, either loading *modules* on HPC system, or manually compiled. For instance, on the HPC Bura, the

```
~$ module spider Foo
```

command allows you to inspect all the available modules matching the *Foo* module name. At the time of writing, both **gsl** and **HDF5** libraries are not currently available on the system, so they have to be installed locally in the home directory by the user;

- add a new symbolic name for the target computer system to the *Makefile.systype* file. For instance, on the HPC Bura, this task is performed using the command

```
~$ echo SISTYPE=\"bura\" >> Makefile.systype
```

- create inside the *Makefile* an if-clause in the "define available system" section, in which two files from the *buildsystem/* subdirectory are included. Those files define path names to the libraries, i.e. either if not in default locations or not configurable through module-environment (e.g. *Makefile.path.<platform_name>*), and that specify the compiler and its flags (e.g. *Makefile.comp.<compiler_name>*). For instance, on the HPC Bura, the "define available system" section of the *Makefile* is edited as follows:

```
#########################
#define available systems#
#########################

ifeq ($(SYSTYPE),"bura")
   include buildsystem/Makefile.path.default
   include buildsystem/Makefile.comp.gcc
endif
```

where the *buildsystem/Makefile.path.default* file specifies the locations of the libraries *header* files and *binaries*) installed locally in the user home directory required by the **GADGET4** code (e.g. **gsl** and **HDF5** libraries), while the *buildsystem/-Makefile.comp.gcc* file specifies the compiler and all the flags of the compilation process.

The *Makefile* should not be modified in any significant way to ensure code portability, except for adding additional source files to the code.

4.4.3 Running the GADGET4 code

To start a simulation, the executable has to be invoked with the following command:

```
mpirun -np <Ntasks> <executable> <paramfile>
```

This will start the simulation with *Ntasks* MPI processes, and with simulation parameters, as specified in the parameter file. The code is able to automatically detect groups of MPI processes running on the same node, and it allows a shared-memory communication scheme for their data. If more than one node is in use, at least one MPI process on each node is set aside for asynchronously serving incoming communication requests from other nodes. As a consequence, multi-node jobs must have at least two MPI processes on each node.

For example, on the HPC Bura where the batch system SLURM is in use (see Chapter 2), a batch script similar to the code 4.4 could be used to start a simulation using two nodes with 24 MPI processes each (i.e. using 48 cores in total). The code uses 46 MPI processes to perform the computational work because on each node one process is devoted purely to communication purposes.

Algorithm 4.4 A SLURM script used to start GADGET4 code.

```
#!/bin/bash

#SBATCH --job-name=gadget4
#SBATCH --nodes=2
#SBATCH --time=00:15:00
#SBATCH --ntasks-per-node=24
#SBATCH --output=gadget4-%j.out

echo
echo "Running Gadget4 on hosts: ${SLURM_NODELIST}"
echo "Running on ${SLURM_NNODES} nodes"
echo "Running using ${SLURM_NPROCS} processors"
echo

SECONDS=0

mpirun -np ${SLURM_NTASKS} ./Gadget4 ./paramfile.txt &> log.txt

TIME=$(($SECONDS))

echo "Execution time is ${TIME} seconds"
```

A running simulation can be interrupted after every timestep in two ways. The first requires specifying the CPU-time limit value in the parameter file (*TimeLimitCPU* parameter sets the wallclock time limit for the current execution of the code, in seconds), the code will interrupt itself automatically before the CPU-time limit is reached, and write a set of restart-files, so that the simulation can be resumed later on. The second way allows interrupting the code manually by creating a file named *stop* (it can be empty and created using the *$ touch stop* command) in the output directory of the simulation; the code will then write the restart-files after the current timestep has been completed. The latest way to stop a running simulation is at compile time through the **STOP_AFTER_STEP** macro in the *Config.sh* file, setting the end of a simulation after the specified time step. This is meant to simplify performance and scalability tests.

4.4.4 Parallelization options

The *Config.sh* file allows to switch on/off parallelization options, among which:

IMPOSE_PINNING:
if this macro is switched on the code pins MPI processes to cores taking into account the processor topology. This feature requires the **hwloc** library [Broquedis et al., 2010] installed (hwloc library is distributed under the BSD license.). Note that many MPI libraries nowadays in any cases enable pinning by default, or can be asked to arrange for the pinning via options to the MPI start-up command.

IMPOSE_PINNING_OVERRIDE_MODE:
if this macro is switched on then IMPOSE_PINNING is assumed even in the case the MPI start-up has already been establishing pinning.

ENABLE_HEALTHTEST:
if this macro is switched on then the code tries to figure out whether all CPU cores are freely available and what is their execution speed. The MPI bandwidth inter-node and intra-nodes is tested as well.

4.4.5 Parameterfile

The parameterfile must be specified whenever the code is started. Each parameter value is set up by specifying a keyword (string) followed by either a string or a numerical value (float or integer), separated by (an arbitrary amount of) whitespaces. Each keyword needs to be specified once, is type-sensitive, and the order in which they are specified is arbitrary. Lines with a leading % or # are skipped.

In the following, only the keywords needed to run a pure N-body simulation (i.e. without hydrodynamics) using the tree algorithm are discussed:

OutputDir */scratch/n-body_simulation*
This is the pathname of the directory that holds all the output generated by the simulation (snapshot files, restart files, diagnostic files). The folder should exist before the start of the simulation.

SnapFormat 2
A flag that specifies the file format to be used for writing snapshot files. Select 2 for a binary format available from GADGET-2 onwards.

InitCondFile *../CI/Aq-C6-dm*
This sets the filename of the initial conditions to be read in at start-up, and they can be distributed into several files.

TimeLimitCPU 86400
This is the wall clock time limit for the current execution of the code, in seconds. The run will automatically interrupt itself and write restart files if 85% of this time has elapsed. Note that this time refers to the wall-clock time on one processor only, and not to the total CPU time consumed by the code (that it is obtained by multiplying the total number of cores used by the run).

CpuTimeBetRestartFile 7200
This is the maximum amount of wall-clock time, in seconds, that may elapse before the code writes a new set of restart files for regular checkpoint.

MaxMemSize 2048
This value gives the maximum amount of memory (in MByte) the code is allowed to use per MPI process. It is therefore a good idea to set its value to something close to the amount of physical memory that can be used per MPI process on the target compute nodes. The possibly memory-per-core allocated through a workload manager (e.g. in SLURM script through the option *#SBATCH –mem-per-cpu=<size>*) should be set accordingly.

TypeOfOpeningCriterion 0
This selects the type of cell-opening criterion used in the tree walks for computing gravitational forces. A value of 0 results in a geometric opening criterion which is primarily governed by the opening angle theta, while 1 selects a relative criterion that tries to limit the absolute truncation error of the multipole expansion for every particle-cell or cell-cell interaction.

ErrTolTheta 0.45
This is the accuracy criterion parameter, i.e. the opening angle theta, of the tree algorithm if the geometric opening criterion (i.e. TypeOfOpeningCriterion=0) is adopted.

ErrTolForceAcc 0.005
This controls the accuracy of the relative cell-opening criterion (if enabled), i.e.

the alpha parameter in Equation 4.6.

ErrTolThetaMax 1.0
When the relative opening criterion is used, the effective opening angle allowed for a node of little mass may grow very large, possibly approaching the convergence radius of the multipole expansion. The parameter sets the maximum allowed geometric opening angle.

4.4.6 The GADGET4 tree algorithm

Geometry of the tree

A modified BH algorithm is implemented in **GADGET4**. The standard implementation employs a fully refined tree (as discussed in Section 4.3.1), where a new node of half the parent's size is created whenever more than one particle falls into the same octant of the node. Now only when a tree leaf node contains more particles than a given threshold then this node is split into sub-octants. As a positive effect, this strategy reduces the depth of the tree, thus the total number of nodes, reducing the memory requirements accordingly.

Multipole expansion

The gravitational potential generated by particles inside a node at point $\mathbf{x}$ is (using for simplicity $G = 1$):

$$\Phi(\mathbf{x}) = -\sum_{j \in node} \frac{m_j}{|\mathbf{x}_j - \mathbf{x}|} \tag{4.1}$$

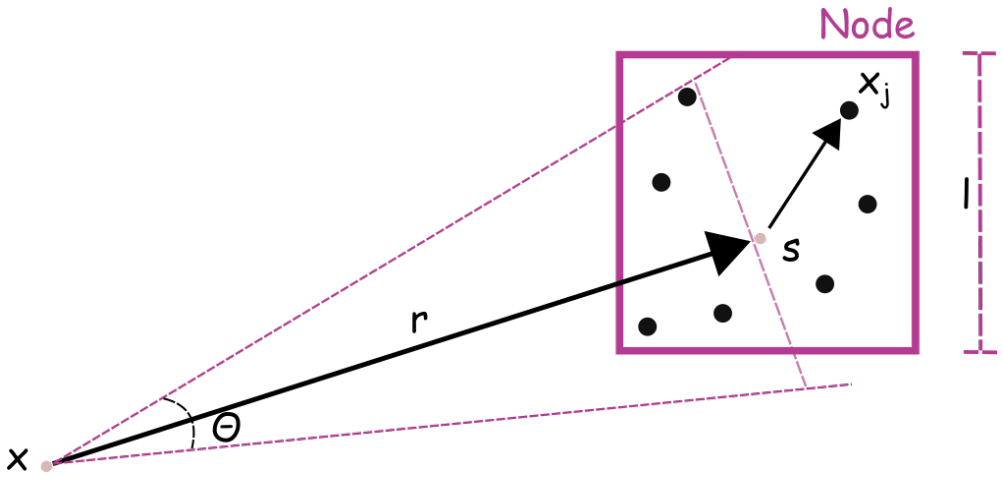


Figure 4.3 Sketch of a tree node and the geometry of the adopted opening angle prescription. The force evaluation occurs at point $\mathbf{x}$, which has a distance r from the node's center-of-mass $\mathbf{s}$, so that it sees the node under an angle of $\theta = l/r$. If this angle θ is lower than the critical opening angle the multipole expansion of the node can be used.

The Taylor expansion up to the order p of the potential around the centre-of-mass $\mathbf{s}$ yields:

$$\Phi(\mathbf{x}) = -\sum_{n=0}^{p} \frac{1}{n!} \mathbf{Q_n} \cdot \mathbf{D_n}(\mathbf{s} - \mathbf{x}) + O(\theta^{p+1}) \tag{4.2}$$

with θ being the angle under which the node is seen, as mentioned in Section 4.3.3.

Here $\mathbf{Q_n}$ are the Cartesian multipole moments:

$$\mathbf{Q_n} \equiv \sum_{j \in node} m_j (\mathbf{x_j} - \mathbf{s})^{(n)} \tag{4.3}$$

and $\mathbf{D_n}(\mathbf{x}) = \nabla^{(n)} \frac{1}{|\mathbf{x}|}$ the derivative tensor. In **GADGET4** the user can select at compile time a selection of $p = 1,2,3,4$ or 5, meaning that the multipole expansion is carried out to dipole, quadrupole, octupole, hexadecupole, or triakontadipole order, respectively.

As a consequence the acceleration exerted on a particle at position $\mathbf{x}$ can be written as

$$\mathbf{a}(\mathbf{x}) = -\nabla\Phi(\mathbf{x}) = -\sum_{n=0}^{p-1} \frac{1}{n!} \mathbf{Q_n} \cdot \mathbf{D_{n+1}}(\mathbf{s} - \mathbf{x}) + O(\theta^p) \tag{4.4}$$

Which order of p is most efficient is typically problem dependent. Using high order reduces force errors at the price of larger memory usage to store more multipole moments for each node.

Tree walk and opening angle criterion

At the stage of the force calculation, the tree is walked top-down, starting at the root node. As mentioned in Section 4.3.3, for deciding whether or not a multipole expansion of the current node is acceptable, the geometric opening criterion, shown in figure 4.3, is employed. The criterion says that a node can be used if

$$\theta > \frac{l}{r} \tag{4.5}$$

where l is the side length of the cell (i.e. cubic node), r is the distance of the target particle to the node's center-of-mass, and finally, θ is a critical angle controlling the accuracy of the force calculation (the **ErrTolTheta** parameter in the **GADGET4** code as explained in Section 4.4.5). Such a parameter tunes the accuracy of the force calculation and the computational cost of the force calculation as well. **GADGET4** allows for an alternative criterion, where a rough approximation of the expected force error is compared with the magnitude of the total force (evaluated from the previous timestep)

$$\frac{M}{r^2} \left(\frac{l}{r}\right)^p < \alpha |\mathbf{a}| \tag{4.6}$$

where α is the **ErrTolForceAcc GADGET4** parameter (see Section 4.4.5).

4.4.7 Domain decomposition in the GADGET4 code

One of the main challenges in contemporary HPC is making full use of the parallel processing power made available by modern computer hardware, which often relies on many independent compute cores. The current release of **GADGET4** expresses parallelism using a relatively traditional approach that relies on distributed (using MPI) and shared memory parallelization, combined with a standard object-oriented programming language C++.

Single compute node is both CPU-time limited and memory limited for large-size simulations of cosmic structures as required by modern cosmology. Hence, parallelization on distributed memory machines is mandatory, and, accordingly, data decomposition strategy plays a key role to scale up the feasible simulation sizes. Data decomposition should make optimal usage of the distributed memory across nodes, avoiding as much as possible redundancy and imbalance. **Load balancing** relies on two main aspects:

- making sure that all processors have an approximately equal amount of work;
- letting the distribution of the domain across processes reflect the structure of the problem so that communications are kept within reason.

The latter item is satisfied by a strategy that tends to preserve the locality of the problem when distributed over a parallel machine: particles' positions that are close together are likely to interact directly (direct force calculation as explained in Section 4.3.3), so they should be handled by the same processor or at least one not too far away (in terms of platform topology), in order either to avoid communication or minimize its overhead. In the case of the BH tree structure, the load balancing becomes a matter of partitioning the leaves of the tree over the processors. The BH tree construction has an intrinsic locality because the subtrees of any non-leaf node are physically close, so they will probably communicate with each other during the task of the force evaluation.

Space-Filling Curves traversals

GADGET4 uses a spatial decomposition in which the simulated volume is divided into disjoint regions subsequently mapped to individual MPI processes. The distributed computational algorithm should not change the semantics of the calculation, precluding, for example, algorithms where the way the domain is cut impacts the geometry of the tree (the force calculation from walking the tree would be affected).

To fulfill these requirements **GADGET4** uses Space-Filling Curves (SFCs). SFCs provide a contiguous mapping from one-dimensional to d-dimensional space, which is used to linearize spatially-distributed data (leaf octants) for partitioning. SFCs have many properties that make them useful for octree traversal:

- they are self-similar (like fractals);
- they visit each octant exactly once;
- they preserve a locality of the mapping onto the d-dimensional hyperspace. In our case, this means that particles that are close in 3D space are typically close to the SFC. So if we map the SFC to a linear ordering of processors we will preserve the locality of the problem.

As 2D examples, two orderings are considered: Morton and Hilbert. The Morton ordering is a simple SFC that traverses the quadrant's children in a z-like pattern in the order I, II, III, IV, as shown in figure 4.4.

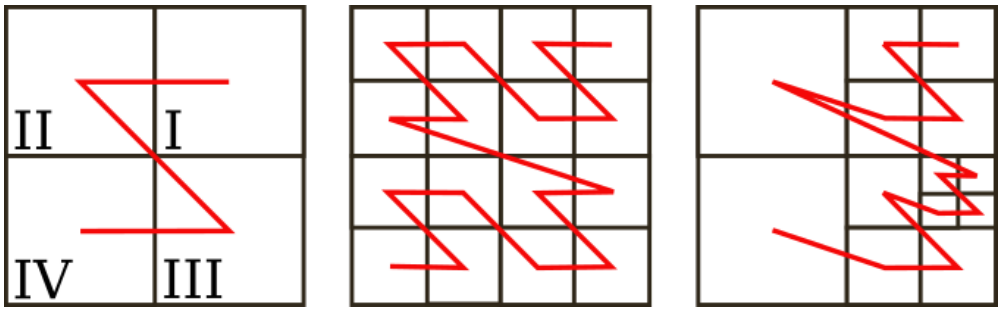


Figure 4.4 Template curve for the 2D Morton ordering (left), its first level of refinement (center), and an adaptive refinement (right).

The pattern at each level of the tree refinement is identical to that used in the first, so neither reflection nor rotation is performed. As a result, spatially distant parts of the domain can occur in its linearization ("jumps" in the linearization), particularly as the

curve passes from quadrant II to quadrant III. The 3D version of the Morton ordering is shown in the left panel of figure 4.6. The template consists of two consecutive z-like curves, where the end of the first is connected to the start of the second.

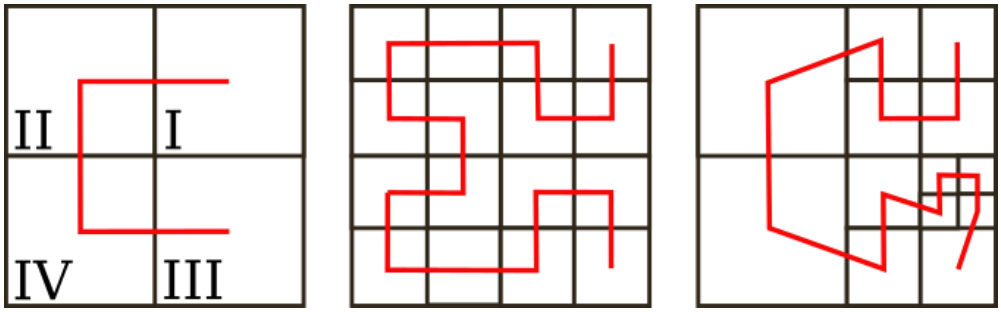


Figure 4.5 Template curve for the 2D Hilbert ordering (left), its first level of refinement (center), and an adaptive refinement (right).

The Hilbert ordering uses the Peano-Hilbert SFC to order the quadrants using a bracket-like template, with extra inversions and rotations to keep quadrants as close as possible to their neighbours, as shown in figure 4.5. The Hilbert ordering added complexity compared to the Morton ordering, which makes the Hilbert ordering harder to construct. The 3D Hilbert template consists of 2D brackets connected at the endpoint with the ordering of the second bracket being opposite to that of the first (right part of figure 4.6).

Fast execution of the octant ordering is essential for an effective load-balancing algorithm. Since the Morton ordering is (relatively) simple and efficient, typically code implementations obtain the Hilbert ordering from it by providing appropriate mapping. The transformations required for the Hilbert ordering are rotations, and more specifically the Hilbert ordering uses 4 orientations in $\mathbb{R}^2$ and 24 in $\mathbb{R}^3$. The mapping is encoded using orientation and ordering lookup tables of dimensions $2^d \times O_d$, where the SFC in $d(=2,3)$ dimensions has the number O_d of unique orientations. A parent octant with orientation i determines the orientation and ordering of its children from row i of each table, $i = 0,1,...,O_d - 1$. Table 4.1 provides the ordering and orientation of the Hilbert indexing in two dimensions.

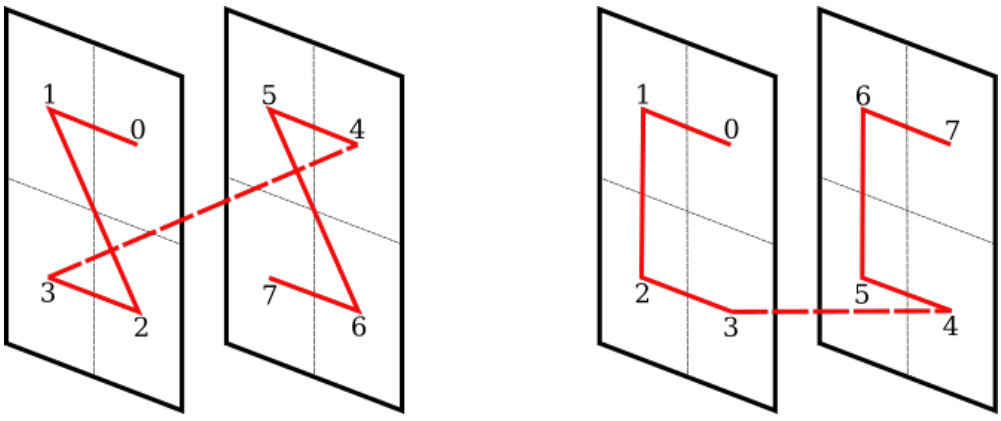


Figure 4.6 Template curve for the 3D Morton ordering (left), and Hilbert ordering (right). The numbers show the traversal order of leaf octants.

For example, the use of the ordering and orientation lookup tables to generate the first two levels of the 2D Hilbert SFC is shown in figure 4.7. The root quadrant

has orientation 0, so the child quadrants at the first level are ordered according to the Morton index sequence provided by row 0 of the ordering Table 4.1: {0 1 3 2}. Analogously, these child quadrants are assigned orientations according to row 0 of the orientation Table 4.1: {1 0 0 2}. The next refinement level uses this orientation to determine the order and orientation of the descendants. For instance, the orientation of quadrant 2 is 2, so row 2 is used in Table 4.1 to determine the child quadrants' order and orientation as {3 1 0 2} and {3 2 2 0}, respectively.

Ordering			
0	1	3	2
0	2	3	1
3	1	0	2
3	2	0	1

Orientation			
1	0	0	2
0	1	1	3
3	2	2	0
2	3	3	1

Table 4.1
Ordering and orientation tables are used to map between Morton and Hilbert SFC in 2D. Row i determines the ordering and orientation of children when a parent with orientation i is refined.

The lookup table defines a string rewriting system with the sequences of characters representing the quadrants that the SFC passes through. Looking at figure 4.7, the starting bracket Hilbert template is represented by the string {0 1 3 2}. On the second refinement level, each character of the string has been replaced with new entries, each of which consists of the old entry concatenated with an appropriate character from a given row of the ordering Table 4.1, using the method previously explained. For instance, quadrant 3 with orientation 0 is replaced by the string {30 31 33 32}. Repeating this for the remaining three quadrants gives the string {00 02 03 01 10 11 13 12 30 31 33 32 23 21 20 22} describing the Hilbert SFC at the second refinement level. This approach works for adaptively-refined curves as well, as in figure 4.5.

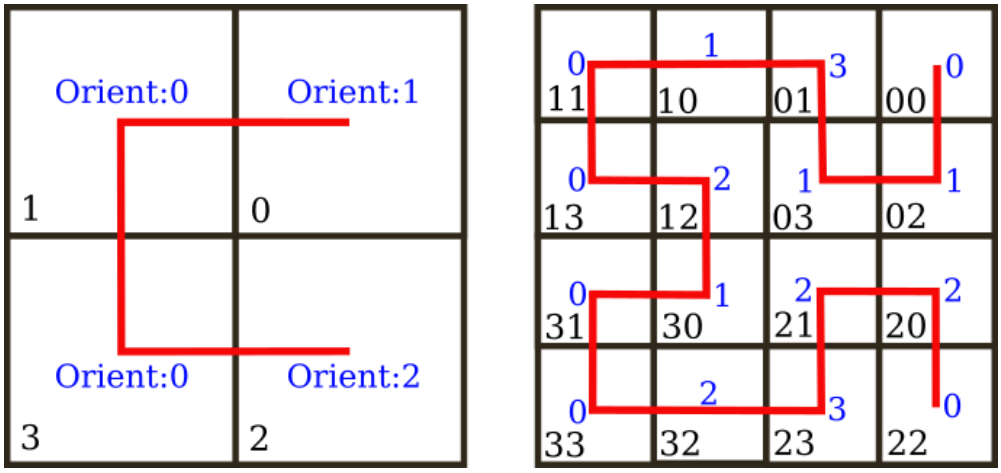


Figure 4.7 The sketch explains the use of ordering and orientation lookup Table 4.1 to generate the first two levels of the 2D Hilbert SFC ordering.

Since the Hilbert SFC preserves better the locality, while the Morton SFC tends to produce irregularly shaped domains, **GADGET4** maps 3D space onto a one-dimensional curve using the Hilbert SFC, carried out with several fast bit-shit operations, and short lookup tables that deal with the different orientations of the fundamental figure. Subsequently, if we simply cut the Hilbert SFC into segments of a certain length, we obtain a domain decomposition that has the property that the spatial domains are simply connected and quite "compact", i.e. low surface-to-volume ratio, which in turn is a highly desirable property for reducing communication costs

with neighbouring domains. If we hence assign an arbitrary segment of the Hilbert SFC to a different partition mapped to a given processor, the corresponding volume is compatible with the node structure of the global BH tree covering the full volume (a group of octans of this tree is effectively assigned to each processor). The described parallelization method does not affect the resulting geometry of the tree, and the results for the tree force become strictly independent of the number of processors used. This approach guarantees that the set of multipole expansions seen by any particle is completely invariant under the domain decomposition. This feature of the domain decomposition does not mean that the results of the force calculation are binary invariant when the number of processes is changed, but differences within floating point round-off error are allowed because mathematical operations are performed in a different order.

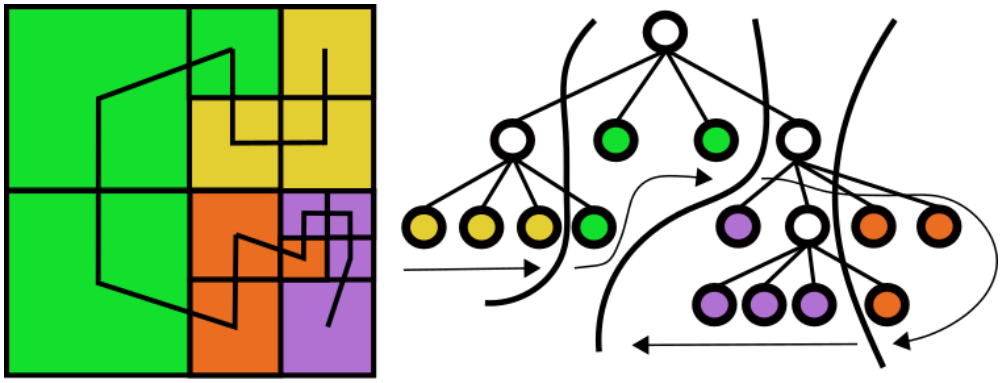


Figure 4.8 Hilbert SFC aimed at building a load-balanced tree adaptively refined. Colors represent the domain partitions assigned to different processors.

The sketch in figure 4.8 shows the relation between the BH quad-tree and a domain decomposition based on a Hilbert SFC. The fiducial Hilbert SFC associated with the simulation volume visits each cell of the adaptively refined tree exactly once. Then the simulation volume is cut into partitions by segmenting this curve into segments of approximately constant work-load (load-balancing problem). The net result of this procedure is that a range of Hilbert keys (one or several consecutive segments chosen such that an approximate work-load balance is obtained, subject to the constraint of a maximum allowed memory imbalance) is assigned to each processor, which defines the domain decomposition and is now used to move the particles to their target processors, as needed. In addition, each processor constructs a *top-level tree* where all nodes at the higher level are represented. However, the local tree has some nodes that consist of *pseudo-particles*, which represent the mass or multipole moments on other processors. These nodes cannot be opened because the corresponding particle data reside on a different processor, but when encountered in the tree walk, the pseudo-particle knows on which processor the actual information resides. Moreover, the sketch illustrates that the problem has a certain locality: the subtrees of any non-leaf node are physically close, so there will probably be communication between them. More importantly, if the domain is refined by another level, the Hilbert SFC can be refined accordingly. The workload can then be redistributed to neighboring processors on the curve, and this process of *load redistributing* still has locality preserved.

A pure optimization strategy, designed to increase the computational speed, is how particles are stored in the memory of each local domain. Particles are ordered in memory according to a finely resolved Hilbert curve, because particles that are adjacent in memory after Hilbert ordering will have close spatial coordinates, so they tend to have quite similar interaction lists of particles. During the force calculation stage, the CPU works on them consecutively, and with such an ordering in many cases, data are already stored in the local cache, which reduces the number of CPU cycles to fetch the data from the (slower) main memory (RAM).

Hybrid parallelization approach

Nowadays the computational performance increase comes from a larger number of compute cores and not from improved single-core performance. In this context, it can be advantageous to combine distributed memory with shared-memory parallelization. A possible choice is to place one MPI process per multi-core compute node, which uses all the amount of memory available on the node, and the other cores are exploited with parallelisation techniques for shared memory, such as OpenMP. In practice, the usage of a hybrid MPI+OpenMP approach can reduce the number of MPI processes to reach a given total memory size and core number. However, the complexity of the algorithms used on **GADGET4**, makes it difficult to reach the same or even higher speed with MPI+OpenMP compared with a pure MPI code exploiting the same number of cores, because only code rearrangements or algorithmic changes are required to make this effective. The software developer then has to face a double challenge of parallelization; an effective distributed algorithm across nodes, which can be further parallelized well at the "local" fine-grained level through a task-based approach.

GADGET4 exploits a new feature of MPI-3, which allows the allocation of shared memory that can be jointly accessed by all the MPI processes (of the same MPI communicator) residing on the same compute node. This feature allows the replacement of common MPI_Send or MPI_Recv operations within a compute node by direct read or write memory accesses like in serial programming, so avoiding MPI communications and synchronization within a shared memory node. In the **GADGET4** code, one MPI process on each node is designed to exclusively handle communications requests. Whenever an MPI process within a compute node wants to access remote memory owned by an MPI rank on a different compute node, it sends the request to the target node's designated communication rank. The latest fetches the data via shared memory access from its compute node, while the target MPI rank is possibly busy doing useful computation and does not have to cooperate for this. In principle, this strategy eliminates the synchronization overhead in shared-memory algorithms, since the designed communication node's rank is not involved in the computation but constantly is waiting for incoming communication requests, which possibly can be answered with minimal latency. Since actual HPC platforms now have more than $\sim$24 cores per compute node (this number is going to increase in upcoming HPC machines), the price to pay for setting aside one core for handling pure communication instead of useful computation does not impact the raw performance. We also note that this strategy of doing MPI-based shared memory programming also i) reduces the memory footprint of the application because some data structures are equal on all MPI-ranks belonging to the same compute node (e.g. the top-level tree), ii) guarantees asynchronous progress of MPI message exchanges while the cores are performing computational work (overlapping of computation and communication as explain in Chapter 2.4.2), iii) reduces the cost for the domain decomposition algorithm and problems of scalability in the MPI software stack.

4.5 Post-processing tools

Post-processing consists of many software tools for analyzing and visualizing volumetric data coming from numerical simulations or observations. Typically they support structured, variable-resolution meshes, unstructured meshes, and discrete or sampled data such as particles, and are focused on driving physically-meaningful inquiry. We briefly describe two of them, publicly available, designed for exploration and visual discovery in particle-based datasets coming from numerical simulations.

4.5.1 Gadgetviewer

This is a program for the visualisation of **GADGET** snapshots (available on GitHub https://github.com/jchelly/gadgetviewer.git). It can read **GADGET4** snapshots (type 1, type 2 and HDF5 formats) and provides an interactive display of the particle distribution, optionally with the capability of colouring any quantity that can be read from the snapshot. There are facilities to pick out populations of particles by various properties (e.g. temperature, density, etc in SPH runs), to follow particles between snapshots, and to make movies. It can be used with other simulation codes which produce **GADGET**-like snapshots.

Command line flags

To get a full list of command line options, run:

```
gadgetviewer --help
```

Reading a snapshot

The name of a snapshot file to read can be specified on the command line or a file can be selected through the *File* menu using the GUI. If the file is part of a multi-file snapshot then the other files will be read as well. Positions, IDs and masses are always read for all particles:

```
gadgetviewer <snapshot>
```

Gadgetviewer allows to read sub-regions from a snapshot using the –region command line flag, e.g.:

```
gadgetviewer --region=x,y,z,r <snapshot>
```

where x,y,z are the coordinates of the center of the selected region, and r is the radius. Since the program reads the entire snapshot and then discards particles outside the region, this can be slow for large snapshots.

Making plots

Gadgetviewer allows to plot histograms of the values of a group of particles or to make a scatterplot of a given property against another (e.g. density vs temperature) through the *Make plot* options menu. Particles in the current sample are shown in red and each set of selected particles is shown in the appropriate colour for that set.

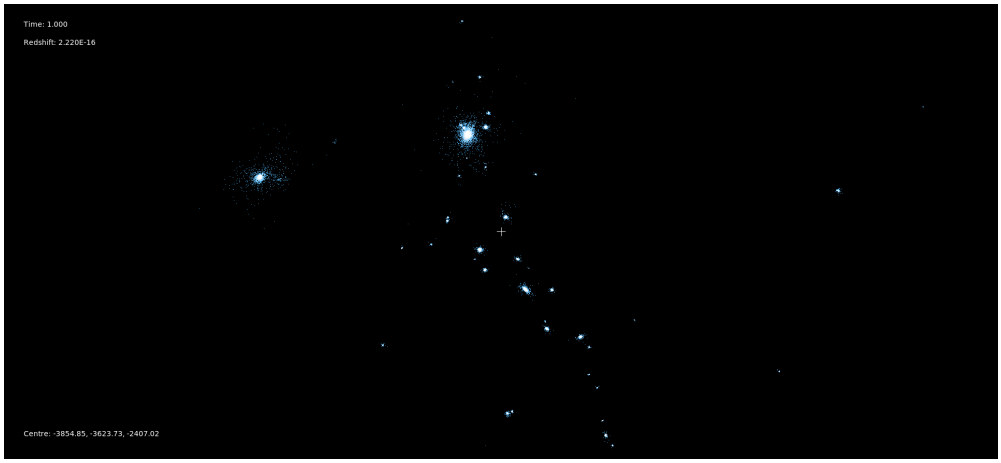


Figure 4.9 Visualization of a **GADGET** snapshot thought the **Gadgetviewer** tool. Star particles are plotted.

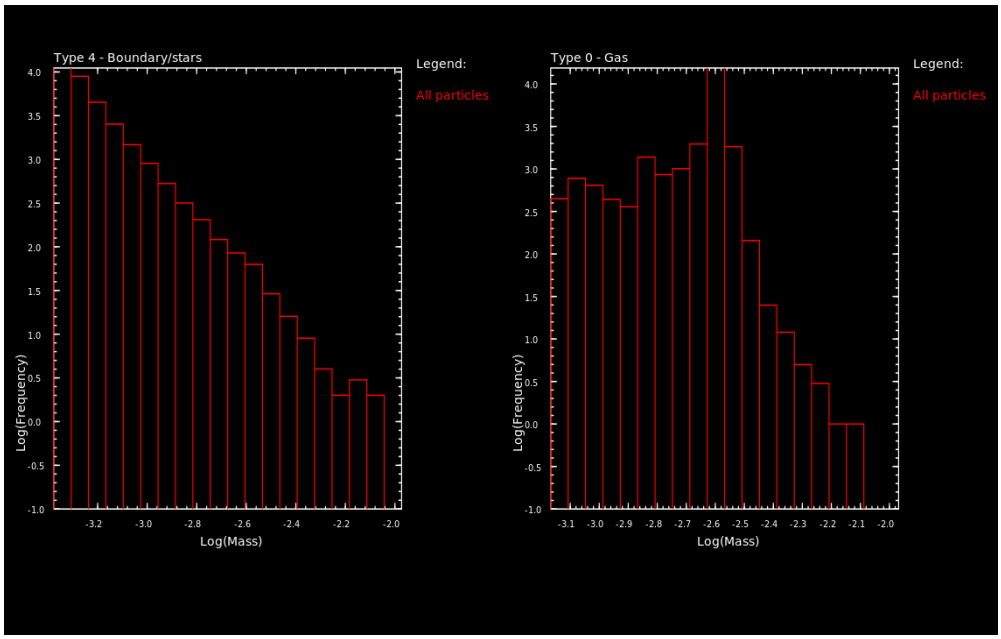


Figure 4.10 Example of histograms generated using **Gadgetviewer**.

4.5.2 Splotch

Splotch [Dolag et al., 2008] is a public available[1] ray-tracer software tool which supports the visualization of cosmological simulations data. The algorithm it relies on is designed to deal with point-like data (**GADGET** snapshot format), performing complex ray-tracing calculations in a fast and effective way. A visual data exploration is a robust approach for rapidly and intuitively inspecting large-scale data sets, e.g. for isolating regions of interest within which to perform time-consuming post-processing analysis or for identifying new features and patterns. Visualization tools can also provide effective means for communicating scientific results not only to researchers but also to members of the general public (e.g outreach and dissemination).

Splotch relies on an effective mix of the **OpenMP** and **MPI** parallel programming paradigms, and can exploit GPU accelerators, which widely populate nowadays HPC

[1]Splotch repository: https://github.com/splotchviz/splotch

architectures, using the **CUDA** programming language (**CUDA** implementation in **Splotch** is described in detail in Rivi et al. [2014]).

Splotch overview

The **Splotch** code is written in C++ and is self-contained with no dependencies from external libraries (apart from **OpenMP**, MPI, and specific file formats, e.g. HDF5). The main stages of the code are briefly summarized below:

1. *Data loading* - many readers are available supporting custom file formats (including the **GADGET** file format). At least the particle's coordinates are required (three scalars), optionally along with physical quantities (e.g. mass, density, temperature) and other geometric quantities (e.g. smoothing length).
2. *Processing and Rasterization* - firstly, particle coordinates are roto-translated and projected according to camera settings, which sets the line of sight and the width of the field of view. Then, active particles that lie within the scene are identified and assigned with RGB colour component; this stage is referred to as *Rasterization*.
3. *Rendering* - The contribution of the active particles to the final rendering (the process of generating a synthetic image) is calculated by solving the (simplified) radiative transfer equation along lines of sight originating from each pixel:

$$\frac{d\mathbf{I}(\mathbf{x})}{ds} = (\mathbf{E_p} - \mathbf{A_p}\mathbf{I}(\mathbf{x}))\rho_p(\mathbf{x}), \tag{4.7}$$

where $\mathbf{I}(\mathbf{x})$ is the radiation intensity at position $\mathbf{x}$, s represents a coordinate along the line of sight, $\mathbf{E_p}$ and $\mathbf{A_p}$ are the coefficients of emission and absorption of the particle p and $\rho_p(\mathbf{x})$ is the tracer transported by the particle (e.g. mass, density, temperature, or energy) described with a Gaussian distribution:

$$\rho_p(\mathbf{x}) = \rho_{0,p} \exp\left(-\|\mathbf{x} - \mathbf{x_p}\|^2/\sigma_p^2\right), \tag{4.8}$$

where $\mathbf{x_p}$ denotes particle coordinates, and σ_p is the particle smoothing length. In practice the distribution is clipped to zero at a given distance $\Lambda = \lambda \cdot \sigma_p$, so that any ray passing at distance larger than Λ is unaffected by the physical quantity ρ_p. The Equation 4.7 is further simplified assuming that $\mathbf{E_p} = \mathbf{A_p}$, so that the solution does not depend on the particles integration order along the line of sight, thus simplifying the parallel design of the algorithm. The assumption typically produces visually appealing images (e.g. 4.11). Equation 4.7 is solved for each colour component (R, G, B) separately but concurrently in parallel. If the ρ_p quantity is a scalar (e.g. mass, temperature, or density), then can be mapped to RGB components via look-up tables (user-defined colour palettes), while when ρ_p is a vector the mapping to RGB components is performed using vector components, e.g. $\rho_p^R = v_x$, $\rho_p^G = v_y$, and $\rho_p^B = v_z$.

Through MPI the code distributes chunks of particles among different processors, each performing a serial computation and producing a partial rendering, subsequently gathered by the root processor which composes the final renderings (all the partial contributions are merged by means of a collective reduction operation producing the final image). The *Rasterization* stage is parallelized in shared memory exploiting multiple **OpenMP** threads working on a different bunch of particles. The *Rendering* stage is more complex because an efficient *one-particle-per-thread* approach without race conditions is not possible and memory usage must be managed carefully. This is

due to the fact that the *Rasterization* stage assigns to each particle its projected radius $R_0 =< r(p) >$

$$r(p) = A(p)\frac{\lambda\sigma_p}{S_{box}}N_{pix} \tag{4.9}$$

where $A(p)$ is the transformation to screen coordinates, λ and σ_p are defined in Equation 4.8, S_{box} represents a normalization factor measuring the size of the simulated box containing all the particles, and N_{pix} is the horizontal (or vertical) image resolution. As a result of Equation 4.9 the typical case scenario is a cluster of particles with different radii each influencing common screen pixels. The dependency from $r(p)$ is the source of two major difficulties for GPU implementation. Firstly, two **CUDA**-threads acting on different particles could try to update the same screen coordinates causing a race condition because of concurrent accesses to the same memory location and thus leading to erroneous results. Secondly, as different particles have a different projected radius R_0 (i.e. they affect a different number of pixels), it is hard to achieve optimal workload balancing among **CUDA**-threads. Customised solutions have been adopted to circumvent the aforementioned problems while avoiding paying large performance penalties [Rivi et al., 2014].

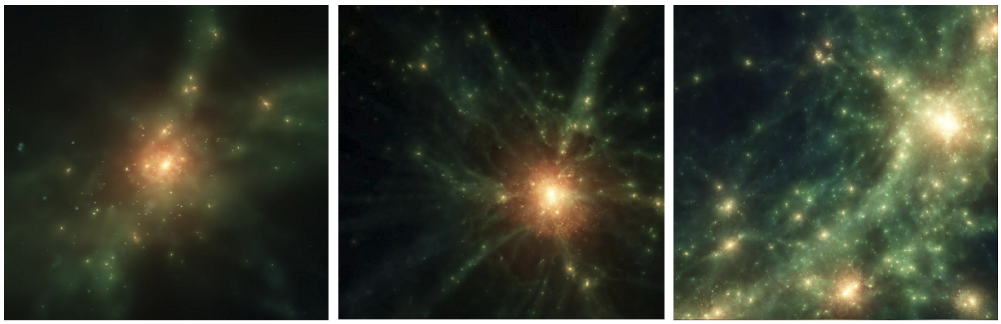


Figure 4.11 Example of visualizations of small (left), medium(middle) and large (right) data sets. Images from Jin et al. [2010].

5. Evolutionary Computation with JGEA

Evolutionary computation
Evolutionary computation software
JGEA structure and components
Experimental evaluation: two case studies
Concluding remarks

5.1 Evolutionary computation

University of Trieste

Evolutionary Computation (EC) is one of the main families of approaches to solving optimization problems in artificial intelligence (AI), taking inspiration from the Darwinian Theory of Evolution as introduced in [Darwin, 1859]. In order to discuss EC, it is necessary to define what an *optimization problem* is in this setting. In a quite general form, only two components are needed:

- A set S of possible solutions. This set need not be finite or have any specific structure, but only provide "solutions". Here a solution can be more or less anything: a permutation of nodes in a graph, a production schedule, the shape of an antenna, a controller for a robot, etc.
- A way of comparing solutions, generally in the form of a *fitness function* $f : S \to \mathbb{R}$ mapping each solution to a real number expressing the *quality* of a solution. This quality can either be minimized (e.g. if it is a prediction error) or maximized (e.g. if it represents the amounts of goods produced by a specific schedule).

Given the exceptionally wide class of optimization problems and the generality of the final objective, i.e. finding either $\arg\max_{s \in S} f(s)$ of $\arg\min_{s \in S} f(s)$, it is important to remark that there exists no general efficient solution to finding the optimal solution to a problem. For some problems performing an exhaustive search is possible or there exist exact efficient algorithms to find a solution (e.g. if the problem can be formulated as a linear programming problem). In such cases, there is no need to resort to AI techniques in general and EC in particular. For some problems, however, there is no known efficient exact or approximate algorithm and even the problem itself might have a fitness function that is a "black box". If that is the case and good - but maybe not optimal - solutions can be acceptable, then EC provides a general way of solving a large class of optimization problems while requiring the practitioner to specify only the problem, leaving the solution strategy to one of the many EC algorithms.

To make EC more concrete, in this chapter we will introduce the basic notions of evolutionary algorithms (EA) using as an example one of the most used EA, namely *Genetic Algorithms* (GA). For a more in-depth introduction to EA and metaheuristics in general, we refer the reader to the introductory book by Sean Luke [Luke, 2013]. We will also introduce the reader to the current landscape of EC libraries before introducing the actual focus of the chapter: the *Java General Evolutionary Algorithm* (**JGEA**) framework (`https://github.com/ericmedvet/jgea/releases/tag/v2.0.2`), which was introduced in [Medvet et al., 2022], of which this chapter is an extended version. We assume that the reader has a passing familiarity with java or with the concepts of object-oriented programming.

5.1.1 Genetic algorithms

Introduced by John Holland in the Seventies [Holland, 1975], GA, at least in their simplest form, are based on the idea of solving optimization problems where solutions can be encoded as a binary string of fixed length n.

Genotype and phenotype

In each solution, we will distinguish the *phenotype* and the *genotype*. The genotype is the representation of the solutions - in this case a binary string of fixed length. The

phenotype is the actual solution being represented. For example, it is possible to encode a subset B of a given finite set A as a binary string representing the elements of B that a present in A. While the set B is the phenotype, the binary string representing it will be the genotype. The operations that we are going to define operate only on the genotype or the phenotype, usually never both together.

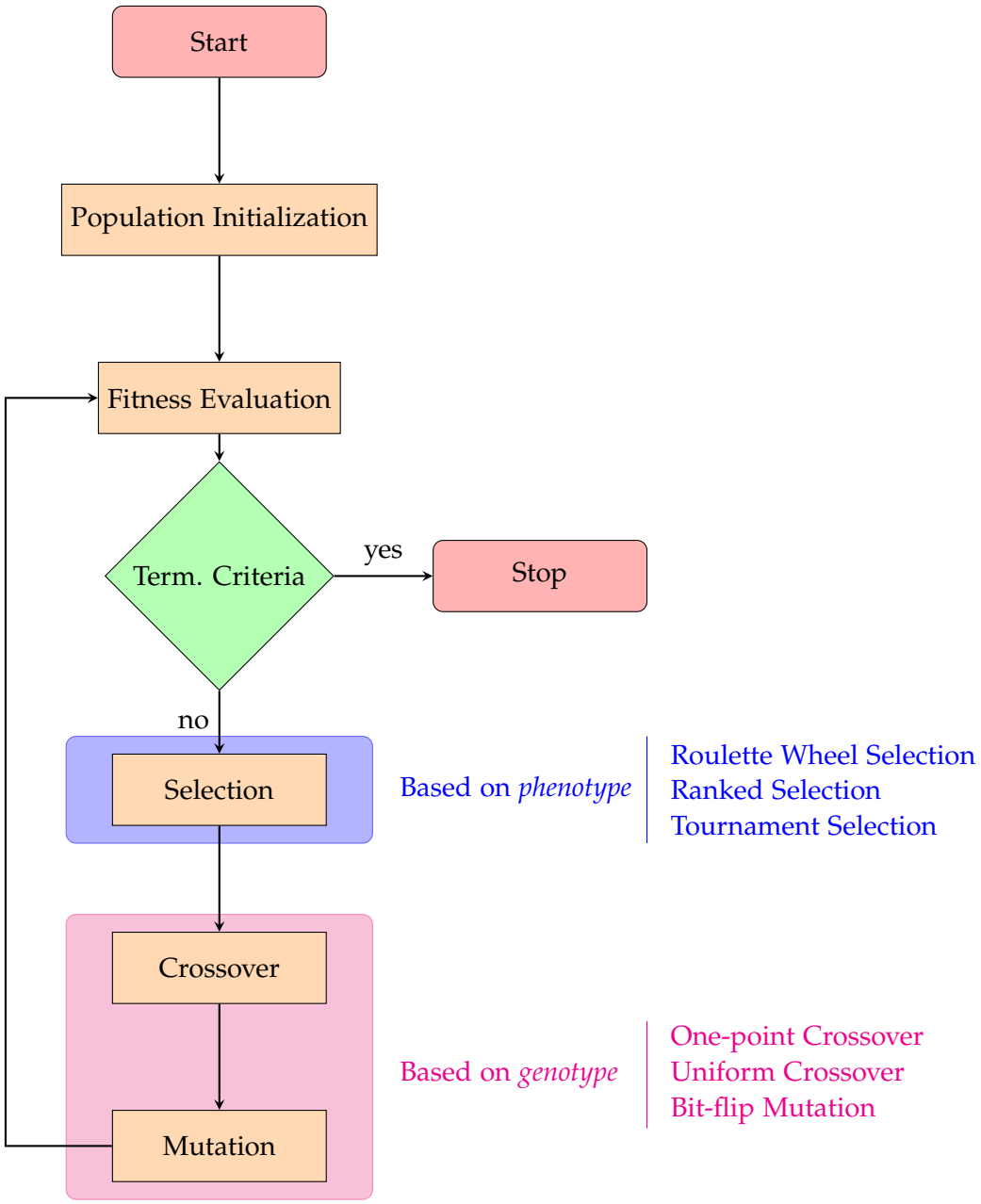


Figure 5.1 The standard evolution cycle employed by most GA. The actions operating on the phenotype are generally shared among different representations, while the ones based on the genotype are specific for each representation. For each (general) operation some of the most commonly used GA operations are presented.

The evolution cycle

Like all EA, which are population-based optimization methods, GA employs a multiset P, i.e. a set with multiplicities, of solutions which is iteratively improved with time.

Such a set is called a *population* and each solution is called an *individual*, following the metaphor of natural evolution. This iterative improvement is performed using a collection of operations employed one after the other:

1. **Fitness Evaluation**. This step, sometimes partially overlapping with the next one of selection, consists in computing, for each element of P, the quality of the represented solution (i.e. of the phenotype). This is usually performed by evaluating a problem-specific function $f\colon \{0,1\}^n \to \mathbb{R}$ on each of the elements of P.
2. **Selection**. Mimicking natural selection, this step consists in a resampling (with reinsertion) of P according to the fitness of the individuals, with individuals with better fitness having a higher probability of being selected. Different selection methods provide different probability distributions, usually in an implicit way. This operation, making use only of the fitness of an individual, clearly operates only on the phenotype.
3. **Crossover**. Also called recombination, this operation mimics the reproduction of pairs of individuals that happens for natural populations. Here, the binary strings of two solutions that were selected in the previous step have some of their components exchanged to generate new solutions. Since this operation works directly on the solutions' representation, it only deals with the genotype of a solution.
4. **Mutation**. Similarly to DNA mutations happening in the real world, some of the bits of a solution can be flipped with a small probability. Mutation, thus, ensures that the pattern of bits that are absent in a population can reappear during the mutation process. As with crossover, also mutation deals only with the genotype of an individual.

A GA can thus be described as shown in figure 5.1, where an initial population of binary strings of length n is initially generated by sampling uniformly at random the space $\{0,1\}^n$, then the operations of fitness evaluation, selection, crossover, and mutation are iteratively applied until some termination criteria are not satisfied. Each such iteration is called a *generation*. Finally, the termination criteria are usually in the form of a limit in the number of fitness evaluations, in reaching a certain fitness threshold or having no increase in the best individual in the population for a certain number of iterations.

Selections

As stated above, selection algorithms operates based on the fitness of the individuals and ignore the actual genotypes of the solutions. One of the simplest selections is *Roulette-wheel selection*, where each individual is assigned a probability of being selected that is proportional to the fitness of that individual. Assuming that the fitness f needs to be maximized, then the probability of individual x to be selected in population P is given by $\frac{f(x)}{\sum_{y \in P} f(y)}$. Since one individual with a very large fitness can actually "dominate" the selection, it is usually preferable to use the ranking of the solutions instead of using the fitness value directly, as done in the *ranked selection*. One of the most used kinds of crossover is *tournament crossover*, combining a simple implementation with the advantages of ranked selection. The idea is to extract (with reinsertion) $t \in \mathbb{N}$ individuals from the population and then select the one with the best fitness. By changing t, i.e. the *tournament size*, it is possible to change the selection pressure, with larger values of t making the selection of less fit individuals more unlikely.

Crossovers

Crossover is based on the idea that two different solutions can evolve good - or even optimal - subsolution independently as long as there is a process allowing them to combine them. Crossover starts with two individuals $x = x_1, \dots x_n$ and $y = y_1, \dots, y_n$ and combines the bits of the two. The classical crossover for GA is *one-point crossover*, where a random number $k \in \{1, \dots, n\}$ is generated and two new solutions are generated by swapping the bits of x and y after position k:

$$x' = x_1, \dots, x_k, y_{k+1}, \dots, y_n \qquad y' = y_1, \dots, y_k, x_{k+1}, \dots, x_n$$

While one-point crossover mimics well the idea of splitting and recombining individuals, it is dependant on the order in which the bits are used in the representation, e.g. bits in position 1 and n will almost always be split apart by crossover, while 1 and 2 will almost always be kept together. *Uniform crossover* is usually employed to avoid the problems of one-point crossover. There, each bit between the two parents is exchanged independently from all other bits with a probability of 0.5.

Mutations

In GA mutation is usually performed as *bit-flip mutation*. Given an individual $x = x_1, \dots x_n$, each bit is individually flipped with a probability p_m, called the *mutation probability* which is usually small (a common default value is $p_m = 1/n$). The effect of mutation is to allow the search process of GA to be able to reach each possible solution in the search space. Since mutation changes a solution in a random way - not even recombining two possibly good solutions like crossover - it is important to keep the mutation probability small: a value that is too high would make GA behave similarly to random search.

From GA to EA: why a general software framework

As it is possible to observe, even by selecting one standard EA algorithm, namely GA, there are multiple "moving parts" to control, from the different operators to employ, to the way the fitness is defined. With more complex representations and more complex EA, the support of a library able to allow easy implementation of different EC techniques is essential. Thus, the EC framework should be target-oriented, focusing on the exigencies of the community. As such, it should be both *solid* and *extensible*. In fact, people who resort to EC as a mere tool for solving problems aim at solidity, i.e. they require a simple and usable system, which gives some guarantees and already has built-in algorithms to be used off-the-rack. Conversely, EC researchers demand an extensible framework, which allows the implementation of new algorithms or representations leveraging lower-level abstractions, and gives the possibility to test them on already written benchmarks, without incurring divergent change due to the additions. Both requirements call for clean and solid modeling of core concepts in EC, based on advanced design and programming techniques, and taking advantage of state-of-the-art design patterns. In addition to the ability to easily define the problem to be solved and the EA to use, most EC techniques can easily be implemented in a parallel and distributed manner. For example, fitness evaluation, selection, crossover, and mutation can all be performed independently, having only to reconstruct the population at the end of each generation. A modern EC framework must be able to extract this parallel component of existing EA and run them on multiple execution units or even machines. As we are going to see, those are not properties that are easy to find in EC software, however, **JGEA** is a modern EC framework that satisfies most of the requirements of a modern, extensible, and efficient EC framework.

5.2 Evolutionary computation software

When researchers and practitioners start to search for which EC software to employ, they find that the landscape is largely different from other areas of AI research. For neural networks libraries like PyTorch [Paszke et al., 2019] or Tensorflow [Abadi et al., 2015] are used in most of the existing software, but such widespread and well-known libraries are non-existing in the area of EC.

Software for experimenting with EC was traditionally hand-crafted for individual problems, without aiming at generality or re-usability, thus hampering the ability of EA to be widely employed. Already in one of the most successful early frameworks, lil-gp [Zongker et al., 1995], significant modifications were needed for application to many different domains like, for example, to RoboCup [Luke, 2017]. The first ideas regarding a more general approach to the design and implementation of software systems were put forward, and the current of thought was started by works proposing ideas for more general frameworks for specific EA, like Genetic Programming (GP) [Keith and Martin, 1994, Cona, 1995] and memetic algorithms [Krasnogor and Smith, 2000]. Some years later, the work of Gagné and Parizeau [2006] formalized the design principles to be followed when modeling an EC general framework, including discussions about representations, parameters management, and reconfigurability.

Good intentions notwithstanding, the first examples of frameworks in the field of EC were mostly focused on meeting a subset of the proposed requirements, inevitably leaving some desirable features behind. Among them, we find some interesting yet sectoral examples written in C, like the GP-oriented lil-gp [Zongker et al., 1995], or PGAPack [Levine, 1996] and GAUL [Adcock, 2009], both focusing on Genetic Algorithms (GAs). Concerning the general EC frameworks, instead, C++ and Java seemed to be the preferred languages. Notably, the objectives of extensibility and ease of use were rarely achieved altogether. In fact, on one side there were modular frameworks, designed with open architectures, like the C++ Open BEAGLE [Gagné and Parizeau, 2002] or the Java ECJ [Luke et al., 2006], JEAF [Caamaño et al., 2010], or JCLEC [Ventura et al., 2008]. On the other side, there were foolproof systems, intended for the general public, such as the EO [Keijzer et al., 2001], written in C++, or its Java counterpart, Evolvica [Rummler and Scarbata, 2001, Rummler, 2007], both featuring a GUI.

Near all of the listed frameworks have been abandoned as of today, with few exceptions, like JCLEC [Ramírez et al., 2015] or ECJ [Luke, 2017, Scott and Luke, 2019]. Nonetheless, most of the updates were carried out ensuring backward compatibility, thus not including modern language features, or focusing on speed of execution over clarity, becoming cumbersome to use or extend.

In the last decade, there has been a sprout of novel EC frameworks encompassing up-to-date design patterns and programming styles. A vast majority of them were written in Python, due to its growing popularity in the science community, but more exotic languages like R (ecr [Bossek, 2017, 2018]), Julia (EBIC.JL [Renc et al., 2021]), or even JavaScript [Merelo et al., 2014, 2016] were not left out of the game. Concerning the Python ones, some were still rather sectoral, as PyshGP [Pantridge and Spector, 2017], Ponyge2 [Fenton et al., 2017], jMetalPy [Benitez-Hidalgo et al., 2019], or EvoJAX [Tang et al., 2022], but there were also general ones, like DEAP [Fortin et al., 2012], or the more recent LEAP [Coletti et al., 2020]. Various modern EC frameworks were also developed in more traditional languages like the C and Python-based GA-lapagos [Jamieson et al., 2020], Operon C++ [Burlacu et al., 2020], and some Java-written ones. Among the latter, Chips-n-salsa [Cicirello, 2020] and Jenetics [Wilhelmstötter, 2019] are

fairly similar to **JGEA**: as they are both frequently updated and make use of modern Java features. However, unlike **JGEA**, Chips-n-salsa is not suited for GP, while Jenetics does not appear to be explicitly designed for research-related purposes.

5.3 JGEA structure and components

Here we describe the **JGEA** framework, which we designed and developed to address the previously mentioned problems. **JGEA** is written in the Java programming language (namely Java SE 17), which is naturally suited for modeling complex and variegate concepts, thanks to its object-orientation and the presence of adequate syntactic constructs, e.g. generics and interfaces. Moreover, Java is portable and it can also be used in combination with other non-Java tools by virtue of language bindings. **JGEA** is modular and it displays different levels of abstraction, to accustom the needs of researchers planning to extend it without having to re-write new algorithms from scratch. To make **JGEA** accessible to end users, the levels of abstraction go down to the implementation of ready-to-use algorithms. In addition, we provide **JGEA** with benchmarks to ease the testing of new algorithms and with some additional features to optimize the execution of experiments and monitor them. The framework has been in use and updated since 2018. It is currently at version 2.0.2 and available for download at `https://github.com/ericmedvet/jgea/releases/tag/v2.0.2`. **JGEA** has been employed for the experimental evaluation in more than 25 research papers.

The architecture of **JGEA** has been designed to capture the main components of an EA. The goal of the solver is to find better and better solutions to the problem, potentially obtaining an optimal solution. Those high-level concepts are captured in **JGEA** with two interfaces: **Problem** and **Solver**. Starting from those two (general) main components, **JGEA** defines more specific interfaces and implementations for the specific problem being tacked and the specific EA being used as a solver. A fraction of the class structure of **JGEA** is shown in figure 5.2.

5.3.1 Problem

The **Problem** interface should be implemented by any class whose aim is to describe a problem in terms of a set of solution **S** and a partial ordering between solutions given by a **PartialComparator<S>**:

```
1 public interface Problem<S> extends PartialComparator<S> {}
```

This comparison is generally employed to assign a *quality measure* to a solution, like the *fitness* of most EA algorithms. Due to the widespread use of this comparison mechanism, a specific **QualityBasedProblem** interface is part of **JGEA**, adding two functionalities to the **Problem** interface: a function deputy to map a solution to a quality value, and a way of comparing qualities.

```
1 public interface QualityBasedProblem<S, Q> extends Problem<S> {
2   Function<S, Q> qualityFunction();
3   PartialComparator<Q> qualityComparator();
4 }
```

Here, **Q** represents the *quality-* or *fitness-space*.

Note that **qualityComparator()** returns a **PartialComparator<Q>** and not a **Comparator<Q>**, as we do not necessarily want to enforce total ordering between qualities of solutions. For those cases in which we do want to enforce this, we extend **QualityBasedProblem** to a **TotalOrderQualityBasedProblem**. This interface adds a **totalOrder-**

Comparator(), and provides a default implementation for the **qualityComparator()**, where a **PartialComparator** is obtained from the **Comparator**.

```
public interface TotalOrderQualityBasedProblem<S, Q> extends
  QualityBasedProblem<S, Q> {
    Comparator<Q> totalOrderComparator();
    @Override
    default PartialComparator<Q> qualityComparator() { /*...*/ }
}
```

In addition, since oftentimes the quality of a solution is "naturally comparable", i.e. **Q extends Comparable**, we model this extending **TotalOrderQualityBasedProblem** with a **ComparableQualityBasedProblem**.

```
public interface ComparableQualityBasedProblem<S, Q extends
  Comparable<Q>> extends TotalOrderQualityBasedProblem<S, Q> {
    @Override
    default Comparator<Q> totalOrderComparator() {
      return Comparable::compareTo;
    }
}
```

To model more specific classes of problems, it is sufficient to add interfaces extending or classes implementing **Problem** (or one of its subinterfaces, see Section 5.4.1). Among them, we have already included classification problems, symbolic regression problems (including many synthetic functions recommended as benchmarks [White et al., 2013]), multi-objective problems, and various benchmarks as the Ackley function [Ackley, 2012], the Rastrigin function [Törn and Žilinskas, 1989, Mühlenbein et al., 1991], or the K landscapes for GP [Vanneschi et al., 2011].

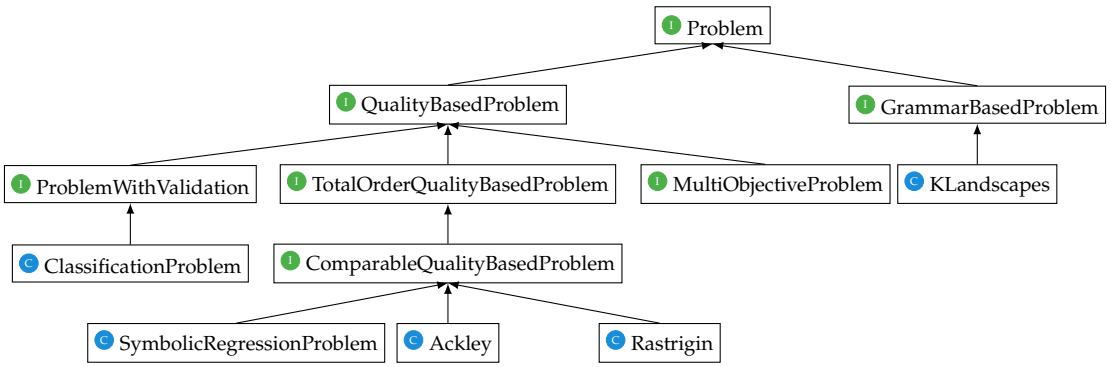


Figure 5.2 Hierarchy of problems.

5.3.2 Solver

A problem can be solved by an implementation of the **Solver** interface, which is responsible for providing the caller with a collection of solutions upon the invocation of its method **solve()**.

```
public interface Solver<P extends Problem<S>, S> {
  Collection<S> solve(
      P problem,
      RandomGenerator random,
      ExecutorService executor
  ) throws SolverException;
}
```

We highlight that, in general, a **Solver** might not be suitable for solving all possible problems. Therefore, we introduce the generic parameter **P** to indicate the subset of problems a **Solver** can tackle.

We also remark that **solve()** takes two additional elements besides the **P** problem: a **RandomGenerator** and **ExecutorService**, since a **Solver** can be non-deterministic and capable of exploiting concurrency. The contract for the **solve()** method is that the passed **RandomGenerator** instance will be used for all the random choices, hence allowing for *repeatability* of the experimentation (*reproducibility*, instead, might not always be guaranteed due to concurrency). Similarly, the contract states that the **ExecutorService** instance will be used for distributing computation (usually, of the fitness of candidate solutions) across different workers of the executor. Population-based optimization methods are naturally suited for exploiting parallel computation (see, e.g. the large taxonomy of parallel methods already developed more than 20 years ago [Nowostawski and Poli, 1999]), and, even though we design **JGEA** aiming at clarity and ease of use, we also take into consideration efficiency.

We designed the **solve()** method of the **Solver** interface in order to model the stateless nature of the solver with respect to its capability, i.e. to solve problems. Namely, since both the **RandomGenerator** and the **ExecutorService** are provided (besides the problem itself) when **solve()** is invoked, different problems may in principle be solved at the same time by the same instance of the solver.

Going more in details, we recall that all relevant EAs share an iterative structure, as the one sketched in algorithm 5.1. We translate it into the **solve()** method of the **IterativeSolver** interface.

Algorithm 5.1 High-level structure of iterative EAs.

```
function solve(...):
  s⃗ ← init(...)
  while!(terminate(...)) do
    s⃗ ← update(...)
  end
  return extractSolutions(s⃗)
end
```

We resort to the *template* pattern [Gagné and Parizeau, 2006] in this interface, thus we do not specify any behavior concerning the initialization, the update, or the termination of the algorithm, and *delegate* those practicalities to the realizations of the **IterativeSolver**. In addition, despite the stateless nature of the **IterativeSolver**, we need a way of keeping track of its execution, e.g. of the population being optimized, hence we introduce the concept of state, **T state**, which is evolved across iterations starting from an initial value. We use a generics parameter since, in principle, different solvers might require to store different structures during their execution. Moreover, the state is used by the **Listener** to monitor the execution (see Section 5.3.4).

```
public interface IterativeSolver<T, P extends Problem<S>, S>
  extends Solver<P, S> {
    default Collection<S> solve(
      P problem,
      RandomGenerator random,
      ExecutorService executor,
      Listener<? super T> listener
    ) throws SolverException {
      T state = init(problem, random, executor);
      listener.listen(state);
      while (!terminate(problem, random, executor, state)) {
```

```
12            update(problem, random, executor, state);
13            listener.listen((T) state.immutableCopy());
14          }
15          listener.done();
16          return extractSolutions(problem, random, executor, state);
17        }
18    }
```

We remark that the structure of algorithm 5.1 which is implemented via the interface **IterativeSolver** is also common to other forms of population-based optimization methods, such as particle swarm optimization or ant colony optimization. Thus, we highlight the ease of extending **JGEA** to accommodate other flavors of metaheuristics starting from already existing core components. Narrowing our view on EAs, we provide one last level of abstraction: the abstract class **AbstractPopulationBasedIterativeSolver**, which implements the interface **IterativeSolver**.

```
1  public abstract class AbstractPopulationBasedIterativeSolver<T
2    extends POSetPopulationState<G, S, Q>, P
3      extends QualityBasedProblem<S, Q>, G, S, Q>
4        implements IterativeSolver<T, P, S> {
5        /*...*/
6  }
```

We model two key concepts related to EAs:

(a) *individuals* - accounting for the genotype-phenotype representation (further detailed in Section 5.3.3)

(b) the *population* - a partially ordered set of individuals, which we store in a custom state, a **POSetPopulationState**.

In addition, thanks to the newly added notions, we can provide concrete implementations of the **init()** method (we initialize the state and sample the initial population, see Section 5.3.3), the **terminate()** method (we check if a given termination condition is verified), and the **extractSolutions()** method (we take the best individual(s) in the population, see Section 5.3.3).

Similarly to what we have seen for the problems, it is possible to extend the abstract class **AbstractPopulationBasedIterativeSolver** to implement specific EAs. In particular, realizations will be characterized by the **update()** method implementation and might require additional customization, e.g. in the state or the initialization (easily achievable through overriding). Thus, we remove the burden of engineering EAs from scratch, providing users with a canvas for specific implementations.

We included in **JGEA** some significant EAs implementations, such as standard GAs (declined in GP [Koza and Poli, 2005], Grammatical Evolution [O'Neill and Ryan, 2001], Hierarchical Grammatical Evolution [Medvet, 2017], Weighted Hierarchical Grammatical Evolution [Bartoli et al., 2018], Context-free Grammar Genetic Programming [Whigham, 1995]), Evolutionary Strategies (ES) [Beyer and Schwefel, 2002], OpenAI ES [Salimans et al., 2017, Nolfi, 2021], CMA-ES [Hansen and Ostermeier, 2001, Hansen, 2016], Map Elites [Mouret and Clune, 2015], Speciated Evolver [Medvet et al., 2021], Diversity Driven Grammar-guided Genetic Programming [Bartoli et al., 2019], Differential Evolution [Storn and Price, 1997], and NSGA-II [Deb et al., 2002].

5.3.3 Individual

We use the notion of *individual*, modeled in the **Individual** record, to capture the genotype-phenotype representation. To this extent, we employ two generic parameters, **G** and **S**, to define the genotype and the phenotype spaces, respectively. In addition,

since we also store the quality (or fitness) of the solution, i.e. of the phenotype encoded by the genotype, within the individual, we also add a generics parameter **Q**.

```
public record Individual<G, S, Q>(
    G genotype,
    S solution,
    Q fitness,
    long fitnessMappingIteration,
    long genotypeBirthIteration
)
```

Creation

To create an instance of **Individual**, we need to:

1. obtain a genotype
2. map it to the corresponding phenotype
3. evaluate the fitness of the candidate solution.

Note that an **Individual** also stores the iteration at which the fitness is evaluated (**fitnessMappingIteration**), and the iteration at which the genotype is obtained (**genotypeBirthIteration**); these values model the "evolutionary age" for the individual in the evolutionary optimization run it belongs to.

A genotype can either be created from scratch or it can be the result of the application of genetic operators on pre-existing genotypes. In the first case, we employ the factory design pattern to *build* random genotypes.

```
public interface Factory<T> {
  List<T> build(int n, RandomGenerator random);
}
```

We provide **JGEA** with some default implementations of factories for the most common genotypes, such as numeric genotypes, bit-strings, or trees.

Concerning genetic operators, we translate the concept into a general interface, extended by two more specific ones, accounting for the mutation and crossover operators.

```
public interface GeneticOperator<G> {
  List<? extends G> apply(
    List<? extends G> parents,
    RandomGenerator random
  );
  int arity();
}
```

We remark that both methods deputy to computing the new genotype, **build()** and **apply()**, take an instance of **RandomGenerator** to ensure reproducibility.

After obtaining a genotype, we map it to the corresponding phenotype with a simple **Function<? super G, ? extends S>**, which can easily be defined on-the-fly, using Java lambda expressions. Last, we compute the fitness of the solution by invoking a **Function<? super S, ? extends Q>**, such as the **qualityFunction()** of a **QualityBasedProblem**.

Selection

Several EAs require selecting individuals for reproduction or survival. To model the selection process in **JGEA** we resort to the **Selector** interface.

```
public interface Selector<T> {
  <K extends T> K select(
    PartiallyOrderedCollection<K> ks,
```

```
    RandomGenerator random
  );
}
```

We provide a few concrete selector implementations, such as the **Tournament**, replicating the tournament selection, or the **First** and **Last**, returning the best or worst individual (or a random one among them, in case of fitness ties). Again, we note the importance of passing an instance of **RandomGenerator** to the **select()** method for reproducibility concerns.

5.3.4 Listener

Even though problems could in principle be solved in-the-void, it is often necessary to track the execution of the solver, extracting and saving information during the run. To this extent, we introduce the last core component of **JGEA**: the **Listener** interface.

```
public interface Listener<E> {
  void listen(E e);
  default void done() {}
}
```

As briefly seen from the code of the **IterativeSolver** in Section 5.3.2, a **Listener** has the duty to monitor, i.e. **listen()** to, the updates of the state during the execution of the **solve()** method.

We delegate the creation of **Listener**s to a **ListenerFactory**, which is used to build "augmented" listeners.

```
public interface ListenerFactory<E, K> {
  Listener<E> build(K k);
}
```

This derives from the need to monitor the execution, either parallel or sequential, of multiple instances of **Solver** solving multiple instances of **Problem**, being able to distinguish individual executions while saving or printing all information on the same target (e.g. the same CSV file for all the evolutionary runs). Moreover, a **Listener** might need additional information (an instance of **K**) to "augment" the results obtained by all invocations of the **listen()** method (e.g. the random seed of the specific run). To this extent, we invoke the **build()** method for each new execution with needed information **K k** passed as an argument, to obtain a properly augmented **Listener** to be assigned to the run. We provide various realizations of the **ListenerFactory** interface, such as the **TabularPrinter**, used to pretty-print useful information on the standard output, or the **CSVPrinter**, employed to save data to a CSV file.

Concerning the information to be extracted from the state, one might be interested in the size of the population, the quality of the best individual, some function of the best individual, and so on. To allow the users to easily define the information they want to extract, and associate a name, and possibly a display format, to it, we introduce the **NamedFunction** interface. Typically, a **List** of **NamedFunction**s is passed to the constructor of a **ListenerFactory**, and each of them is invoked on a state within the **listen()** method to extract the needed information. These constructs make use of modern Java features inspired by functional programming, and favor the achievement of complex behavior, i.e. extracting non-trivial information from an execution, possibly making use of composite functions, in a concise and elegant manner.

A typical example of the construction of a **TabularPrinter** is the following, which prints on the standard output the current iteration, the total number of births, the seconds elapsed since the start of the execution, the fitness of the best individual, and

the distribution of fitness in the population (as a histogram), together with the name of the employed solver (to be passed within a **Map** data structure to the **build()** method), as displayed in figure 5.3.

```
ListenerFactory<POSetPopulationState<?, ?, ?
  extends Double>, Map<String, Object>> listenerFactory =
    new TabularPrinter<>(
      List.of(
        iterations(),
        births(),
        elapsedSeconds(),
        fitness().of(best()),
        hist(8).of(each(fitness())).of(all())
      ),
    List.of(attribute("solver"))
    );
```

It is important to notice that the **listenerFactory** will be able to listen to state updates for **POSetPopulationState**s constrained only to have at least a **Double** as fitness for the individuals. As a consequence, this listener might be used to monitor and track solver runs on different problems, provided they match the kind of state they work on.

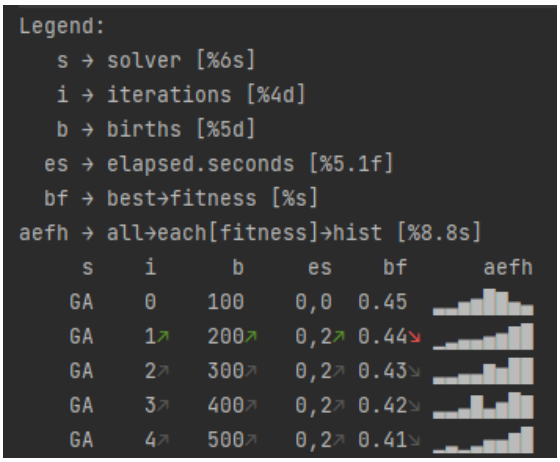


Figure 5.3 Sample output obtained via **TabularPrinter**. For sparing screen space, the names of columns (each being a **NamedFunction**) are automatically abbreviated. Moreover, a graphical indication of the trend for each value is shown in the form of a colored arrow, when appropriate. Through the composition of instances of **NamedFunction**, complex processing of monitored quantities can be performed, as the histogram of fitness values.

5.4 Experimental evaluation: two case studies

Our experimental evaluation of **JGEA** encompasses two case studies, each aimed at testing different, and possibly contrasting, aspects and features of the framework. The first case study, detailed in Section 5.4.1, covers the requirements of ease of use and solidity for end users of an EC framework. In particular, said users might want to adopt EC to solve newly defined problems, even combining the EC framework with external software. The second case study, instead, deals with the extensibility of **JGEA**. This case study, discussed in Section 5.4.2, examines the needs of researchers in the field of EC, who would want to add new algorithms without implementing everything from scratch, being able to test them right away on domain-appropriate benchmarks.

5.4.1 JGEA scalability

In order to assess the suitability of **JGEA** for using EC as a problem-solving tool, we consider the case study of an end user who needs EC to solve an optimization problem. As already mentioned several times, end users who rely on **JGEA** as a mere tool, require a solid and reliable framework, which is handy and easy to understand, and whose computation time efficiently scales with available resources. This unwinds in being able to :

1. simply, yet precisely, define the **Problem** at hand
2. neatly choose and utilize an appropriate **Solver** combined with the right **Listener** to collect the needed information during an execution
3. carry out the experimental evaluation within a reasonable time-frame.

Bridging the first two requirements, there is the concept of representation, as generally users do not evolve solutions to their problem directly, but rather rely on some form of encoding. Therefore, being able to conveniently choose an appropriate solution representation becomes a crucial additional requirement.

We evaluate how **JGEA** meets the listed requirements in sections 5.4.1, 5.4.1, 5.4.1. To avoid being overly general, we focus on the practical setting of evolutionary robotics [Alattas et al., 2019], in which researchers make use of EC to optimize robots-either the controller, the body, or even the sensory apparatus-to achieve a certain task. In particular, we consider the optimization of a neural controller for a class of modular soft robots, similarly to what we have done in [Nadizar et al., 2021].

Problem definition

Given a problem statement written in natural language, e.g. "the goal is to optimize the controller for a robot for the task of locomotion", to formalize it within the **JGEA** framework a user needs to extend the **Problem** interface or any of its subinterfaces. Therefore, they first need to identify the nature of the solution, i.e. determine the solution space **S**, and then they need to decide where to place themselves along the hierarchy of interfaces presented in Section 5.3.1.

In this case, the solution consists of a robot, i.e. an instance of the **Robot** class, which, thanks to its optimized controller, outperforms the others at the locomotion task. This task is already modeled in the employed robotic simulation framework [Medvet et al., 2020], within the **Locomotion** class. Such class is deputy to simulating the locomotion of a robot for t_f simulated seconds and returning its outcome, that is an object storing data about the executed simulation.

```
Outcome outcome = locomotion.apply(robot);
```

Since the user is interested in the quality of solutions (which can be obtained from the **Outcome**) and they can establish a total ordering among them, based on the velocity v_x extracted from the **Outcome**, the proper interface to be implemented is **TotalOrderQualityBasedProblem**. To this extent, it is convenient for the user to define a record, or a class, implementing **TotalOrderQualityBasedProblem**, to avoid creating the **qualityFunction()** and the **totalOrderComparator()** on-the-fly at every invocation.

```
public record LocomotionProblem(
  Function<Robot, Outcome> qualityFunction,
  Comparator<Outcome> totalOrderComparator
) implements TotalOrderQualityBasedProblem<Robot, Outcome> {}
```

Then, the user can simply instantiate the **LocomotionProblem** record, specifying the desired **qualityFunction** and **totalOrderComparator**.

```
Locomotion locomotion = buildLocomotionTask(/*...*/);
Problem problem = new LocomotionProblem(
  robot -> locomotion.apply(robot),
  Comparator.comparing(Outcome::getVelocity).reversed()
);
```

Solver and listener choice and usage

After formally defining the problem, the user needs to *solve* it, i.e. they need to choose and utilize an appropriate **Solver** for the task at hand. We remark that this phase should be as frictionless as possible, even, and especially, for researchers who are not proficient in EC and only use it as a tool. Some basic knowledge of the genotype-phenotype binomial and the main concepts in EC should suffice.

Concerning the genotype-phenotype representation, the user must select an encoding for their solution, i.e. how to represent the robot. This boils down to choosing which parts of the robot are handcrafted and which parts are to be optimized. Here, we consider the case in which the body of the robot is fixed, and the goal is to find the best parameters for the Artificial Neural Network (ANN) that controls it. Therefore, the genotype consists of the list of parameters of the ANN, a **List<Double>**. To obtain a solution, i.e. a robot, from this genotype, the user needs to specify how to build a robot with a controller with those parameters, i.e. they need to specify the genotype-phenotype mapping function.

```
Function<List<Double>, Robot> mappingFunction = list -> {
  MultiLayerPerceptron mlp = new MultiLayerPerceptron(
    /*...*/,
    list //here we specify the parameters of the MLP
  );
  return new Robot(body, new CentralizedSensing(body, mlp));
};
```

Once the genotype space is well-defined (here, **G** is **List<Double>**), the user needs to select an appropriate EA among the available ones. Given the numeric genotype, a simple form of ES is the go-to EA, which is available in **JGEA** in the class **SimpleEvolutionaryStrategy**. Note that the user can employ the chosen EA without having to worry about its internals, as long as they can provide the required constructors parameters, such as the genotype-phenotype mapping function, the **List<Double>** factory, the stop condition, and some numeric parameters.

```
public SimpleEvolutionaryStrategy(
  Function<? super List<Double>, ? extends S> solutionMapper,
  Factory<? extends List<Double>> genotypeFactory,
  int populationSize,
  Predicate<? super State<S, Q>> stopCondition,
  int nOfParents,
  int nOfElites,
  double sigma,
  boolean remap
) { /*...*/ }
```

Even though at this point the user is ready to apply the **Solver** to their **Problem**, they might be interested in monitoring the execution with a **Listener**. For instance, a common need is to keep track of some information during the progress of evolution on a CSV file, which can be easily addressed with the **CSVPrinter**, mentioned in Section 5.3.4.

At this point, all needed elements are defined, and the user can solve their problem, as shown in the following excerpt of code.

```
Solver solver = new SimpleEvolutionaryStrategy(/*...*/);
Listener listener = new CSVPrinter<>(/*...*/).build(/*...*/);
Collection<Robot> solutions = solver.solve(
    problem, random, executor, listener
);
```

Finally, we remark that oftentimes users are interested in performing multiple optimizations, to ensure the generality of their results, regardless of randomness. To this extent, the following code can be used to perform 10 optimizations, printing all the necessary information on the same CSV file.

```
Solver solver = new SimpleEvolutionaryStrategy(/*...*/);
ListenerFactory listenerFactory = new CSVPrinter<>(/*...*/);
for (int i = 0; i < 10; i++) {
  Listener listener = listenerFactory.build(Map.of("seed", i));
  Collection<Robot> solutions = solver.solve(
    problem, new Random(i), executor, listener
  );
}
```

From the resulting CSV file, users can conveniently analyze the experimental data altogether, extracting aggregate information about the progress of evolution and the obtained solutions. For instance, they can effortlessly achieve a report of the progress of solutions fitness along generations, as displayed in figure 5.4.

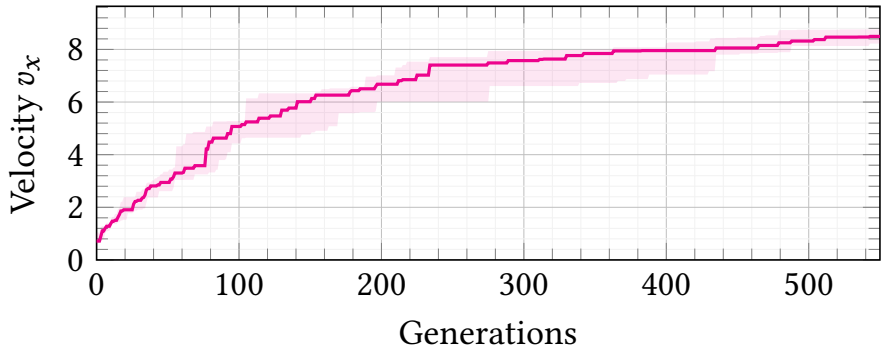


Figure 5.4 The median and interquartile range of the fitness, i.e. the velocity v_x, of the best robot across 10 optimizations during evolution.

Performance and scalability

End users of **JGEA** could suffer from constrained availability of run time or resources, so having well-designed and optimized software becomes not only an added value but also a necessity. Even though EAs themselves are not particularly computationally heavy, some of the operations they need to perform, e.g. fitness evaluation might indeed be more costly from a computational point of view. It is therefore important for the software to take advantage of available resources to distribute the costliest procedures, eventually speeding up the entire process.

Here, we provide a brief experimental assessment of the performance of **JGEA**, to highlight its scalability with the number of available resources. We consider the evolutionary robotics setting described in sections 5.4.1 and 5.4.1, in which the fitness evaluation of each solution-the robot-requires simulating it for a fixed and relatively large amount of time to ensure that the robot is correctly performing the given task: 40 seconds of simulation time takes ≈ 0.8 seconds of (one) core time in our settings. We employ the aforementioned ES, with a population size of $n_{\text{pop}} = 36$. We

emulate the resource constraints by limiting the number of available threads to n_t, and the maximum wall time to t_c (we set the maximum wall time as the termination criterion). We experiment with $n_t \in \{1,2,4,9,18,36\}$ and $t_c = 30$ minutes. We perform the evaluation on a 18 core workstation (Intel Xeon Processor W-2295 from 3.0 to 4.6GHz with 64GB of DDR4 RAM running OpenJDK 17 on Ubuntu 21.10). Since n_t is always smaller than the number of available cores, considering hyper-threading, the threads can constantly execute in parallel. Since this is only an illustrative experiment we perform one execution for each value of n_t, but we remark that normally a user would execute their evaluations multiple times due to randomness.

For each execution, we measure the number of fitness evaluations performed and the velocity of the best robot in the population as a function of elapsed computation time. We report these quantities in the plots of Section 5.5, with a different color for each n_t.

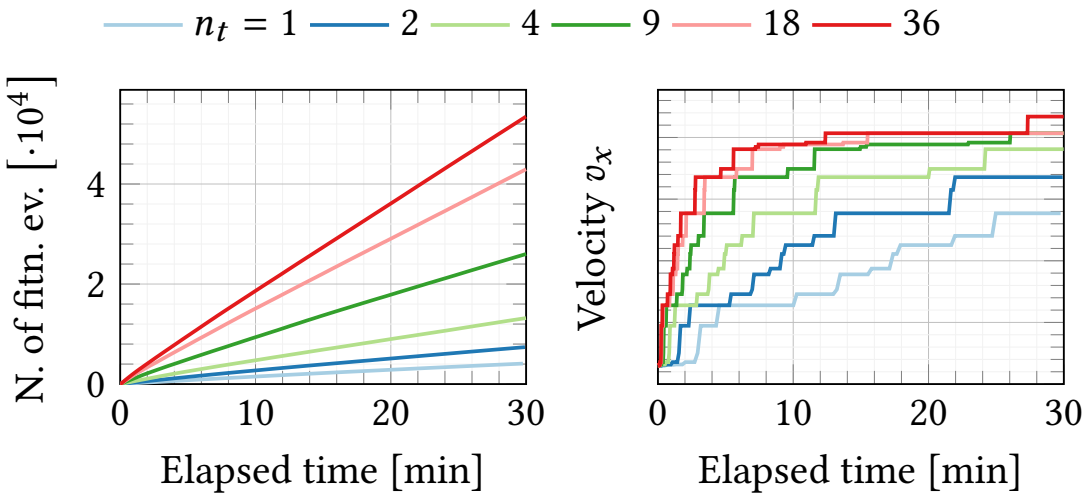


Figure 5.5 The number of fitness evaluations (above) and the fitness v_x of the best individual in the population (below) vs. elapsed time (in minutes).

The results presented in the two figures support our claims concerning the scalability of **JGEA**. Not surprisingly, from the top plot of figure 5.5 we notice a clear linear dependency between the number of fitness evaluations performed and the amount of time, the steepness of which depends on n_t. This is a direct consequence of our implementation of the fitness evaluation process, which is performed in parallel on each of the workers of the **ExecutorService** passed to the **solve()** method. Therefore, by increasing the number of available workers, which corresponds to n_t, we allow more evaluations to be performed simultaneously, thus accelerating the whole optimization process.

Some slightly more interesting results come out of the bottom plot of figure 5.5, where we display the velocity of the best individual as a function of computation time. Even though these outcomes are a direct consequence of the observations drawn above, they are more relevant for practical cases where users have tight time constraints but more available resources and require a good solution quickly.

5.4.2 JGEA extensibility

To the extent of evaluating the flexibility of **JGEA** and its suitability for accommodating new algorithms, we examine the case study of an EC researcher who wants to imple-

ment their newly designed algorithm in the framework. Even though a researcher could, in principle, implement everything from scratch, enclosing their algorithm in an existing framework could prove advantageous from two points of view:

(a) it saves coding time, as there are a gamut of already-defined structures available
(b) it provides a range of ready-to-use benchmarks to test the algorithm on.

Without loss of generality, we consider the specific case of a researcher who implements Map Elites (ME) [Mouret and Clune, 2015] in **JGEA** pretending the algorithm not to be already available in the framework. Our choice fell on ME since it is a quality-diversity algorithm, so it slightly differs from more classical EAs, whose goal is to find only *the best* solution and is, therefore, suitable to prove that **JGEA** is general enough to host various flavors of EAs.

The first necessary step towards the inclusion of ME in **JGEA** encompasses a formal definition of the EA, either in natural language or, preferably, in the form of a pseudo-code. Here, to avoid ambiguities, we consider the situation in which the formalization occurs in pseudo-code, we report it in Algorithm 5.2, but for the sake of clarity, we also provide a brief comment on the code. The algorithm is initialized with the creation of an empty feature map $\vec{\mathcal{M}}$, that is filled with at most n_{pop} individuals. Then, as long as the termination condition is not satisfied, n_{pop} new individuals are created in batch, by mutating randomly selected parents, and added to the map $\vec{\mathcal{M}}$. Finally, the solutions are extracted from the map $\vec{\mathcal{M}}$ and returned.

Algorithm 5.2 ME [Mouret and Clune, 2015] (batched version), with entry point in **mapElites()**, and the inner working of **addToMap()**.

```
1  function mapElites(...):
2    M⃗ ← ∅
3    for c ← 1,...,n_pop do
4      i ← randomIndividual(...)
5      M⃗ ← addToMap(M⃗, i)
6    end
7    while!(terminate(...)) do
8      for c ← 1,...,n_pop do
9        p ← select(M⃗)
10       o ← mutate(p)
11       M⃗ ← addToMap(M⃗, o)
12     end
13   end
14   return extractSolutions(M⃗)
15 end
16 function addToMap(M⃗, i):
17   d⃗ ← extractFeatures(i)
18   if M⃗(d⃗) = ∅ or quality(M⃗(d⃗)) < quality(i) then
19     M⃗(d⃗) ← i
20   end
21   return M⃗
22 end
```

At this point, the researcher needs to frame their EA as a class extending the **AbstractPopulationBasedIterativeSolver**, since it provides basic facilities that can be reused here as, e.g. the initialization and the termination criterion test. To this end, it is convenient to highlight the chunks of ME that match the subroutines of algorithm 5.1 (as already done in algorithm 5.2), namely **init()**, **terminate()**, **update()**, and **extractSolutions()**, since each of them corresponds to a method in the **AbstractPopulationBasedIterativeSolver** abstract class. This step is not only extremely useful to drive design and coding choices, but it also acts as a preliminary testing ground for the proposed EA, helping to identify possible shortcomings. Also, some EAs might

not be directly formalized into the canonical init-update-terminate structure, and this phase is meant for re-framing them.

Then, once the high-level correspondence between pseudo-code and code has been drawn, the researcher needs to start designing the actual **MapElites** class. This phase starts from the signature of the class, where the researcher is inevitably forced to make a decision concerning the allowed values for the generics parameters, **T**, **P**, **G**, **S**, and **Q**. While ME does not pose specific constraints on either **P**, **G**, **S**, or **Q**, its state **T** is required to store additional data besides the usual elements of the **POSetPopulationState**, that is the feature map $\vec{\mathcal{M}}$. Therefore, the research needs to define a custom state, possibly within the **MapElites** class, extending the **POSetPopulationState**, where to store a data structure accounting for $\vec{\mathcal{M}}$, that is **MapOfElites**.

```
public static class State<G, S, Q>
  extends POSetPopulationState<G, S, Q> {
    private final MapOfElites<Individual<G, S, Q>> mapOfElites;
}
```

Next, the researcher will have to identify which additional elements to use in the definition of an instance of the **MapElites** class concerning its super class **AbstractPopulationBasedIterativeSolver**. In particular, ME requires a template for building the **MapOfElites**, i.e. a list of features and a method for extracting the features from an individual, and a mutation operator to build offspring genotypes. We do not cover the details about the data structures employed for representing the listed elements, as these are beyond the discussion.

```
public class MapElites<G, P extends QualityBasedProblem<S, Q>, S, Q>
  extends AbstractPopulationBasedIterativeSolver<
    MapElites.State<G, S, Q>, P, G, S, Q> {
      private final Mutation<G> mutation;
      private final Function<Individual<G, S, Q>,
        List<Double>> featuresExtractor;
      private final List<MapOfElites.Feature> features;
}
```

Last, the researcher must give functionalities to the **MapElites** class. Thanks to the *template* design pattern, the canvas for the **solve()** method is already there, yet some of its subroutines are defined as abstract or need to be overridden for the specific case of ME. Having underlined the correspondence between the functions in algorithm 5.1 and algorithm 5.2, this phase comes relatively straightforward, and mostly consists of raw coding.

```
@Override
public void update(P problem, RandomGenerator random, ExecutorService
  executor, State<G, S, Q> state) throws SolverException {
    List<G> allGenotypes = state.mapOfElites.all().stream().map(
      Individual::genotype).toList();
    Collection<G> offspringGenotypes = IntStream.range(0,
      populationSize).mapToObj(i -> mutation.mutate(allGenotypes.get(
      random.nextInt(allGenotypes.size()))), random)).toList();
    Collection<Individual<G, S, Q>> offspringIndividuals = map(
      offspringGenotypes, List.of(), solutionMapper,
      problem.qualityFunction(), executor, state);
    state.mapOfElites.addAll(offspringIndividuals);
    state.setPopulation(new DAGPartiallyOrderedCollection<>(
      state.mapOfElites.all(), comparator(problem)));
    state.incNOfIterations();
    state.updateElapsedMillis();
}
```

The content of **update()** slavishly follows the prescriptions of algorithm 5.2, with the only addition that the individuals of the **MapOfElites** $\vec{\mathcal{M}}$ are doubled in the **DAG-PartiallyOrderedCollection** of the state to allow the usage of ready-made functions monitoring the population from there.

In conclusion, we observe that adding new EAs to **JGEA** is quite straightforward. In addition, we remark that the implementation process can help researchers to frame their EA in a more understandable manner [Medvet et al., 2021, Nieto-Fuentes and Segura, 2022]. In fact, **JGEA** provides a canvas for the EA implementation, which gently forces a researcher to cast their algorithm into a well-known and clearly defined structure.

5.5 Concluding remarks

In this chapter we presented **JGEA**, an already established and well-developed Java framework for evolutionary computation. We can identify three strong reasons for using **JGEA** for both research on EC and application of evolutionary techniques to real-world problems:

1. **JGEA** provides an expressive way of encoding the common (and not-so-common) kinds of optimization problems that can be found 2in the wild". By decoupling the representation of the solution space and the solution quality it is possible to have a larger range of domains in which **JGEA** can be applied. The framework also allows for easy definition of new evolutionary techniques, as shown in Section 5.4.2.
2. Another reason for using **JGEA** is the scalability of the framework, as shown in Section 5.4.1. **JGEA** allows specifying an executor, thus simply achieving multiple kinds of parallelism.
3. Finally, both for using EAs as optimizers and for performing research on them observability is an important feature. Thanks to the ability to inspect, track, and save information allowed by the **JGEA Listener** interface, it is not necessary to break multiple levels of abstraction to keep track of the state of the evolutionary process.

References

Bibliography

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, i X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous systems. https://www.tensorflow.org/, 2015. Accessed: 01-November-2022.

D. Ackley. *A connectionist machine for genetic hillclimbing*, volume 28. Springer Science & Business Media, 2012.

Adaptive Computing. MOAB Workload Manager. https://adaptivecomputing.com/moab-hpc-suite/, 2021a. Accessed: 01-January-2022.

Adaptive Computing. TORQUE Resource Manager. https://adaptivecomputing.com/cherry-services/torque-resource-manager/, 2021b. Accessed: 01-January-2022.

S. Adcock. Genetic algorithm utility library. *URL http://gaul. sourceforge. net*, 2009.

R. J. Alattas, S. Patel, i T. M. Sobh. Evolutionary modular robotics: Survey and analysis. *Journal of Intelligent & Robotic Systems*, 95(3):815–828, 2019.

J. Barnes i P. Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

A. Bartoli, M. Castelli, i E. Medvet. Weighted hierarchical grammatical evolution. *IEEE transactions on cybernetics*, 50 (2):476–488, 2018.

A. Bartoli, A. De Lorenzo, E. Medvet, i G. Squillero. Multi-level diversity promotion strategies for grammar-guided genetic programming. *Applied Soft Computing*, 83:105599, 2019.

D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, i C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, international conference on parallel processing*, volume 95, pages 11–14, 1995.

A. Benitez-Hidalgo, A. J. Nebro, J. Garcia-Nieto, I. Oregi, i J. Del Ser. jMetalPy: A Python framework for multi-objective optimization with metaheuristics. *Swarm and Evolutionary Computation*, 51:100598, 2019.

H.-G. Beyer i H.-P. Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

J. Bossek. ecr 2.0: a modular framework for evolutionary computation in R. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 1187–1193, 2017.

J. Bossek. Performance assessment of multi-objective evolutionary algorithms with the R package ecr. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1350–1356, 2018.

F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, i R. Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, Feb. 2010. doi: 10.1109/PDP.2010.67. URL https://hal.inria.fr/inria-00429889.

B. Burlacu, G. Kronberger, i M. Kommenda. Operon C++ an efficient genetic programming framework for symbolic regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 1562–1570, 2020.

P. Caamaño, R. Tedín, A. Paz-Lopez, i J. A. Becerra. Jeaf: A java evolutionary algorithm framework. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

A. J. Chorin. Numerical solution of the navier-stokes equations. *Mathematics of computation*, 22(104):745–762, 1968.

V. A. Cicirello. Chips-n-salsa: A java library of customizable, hybridizable, iterative, parallel, stochastic, and self-adaptive local search algorithms. *Journal of Open Source Software*, 5(52), 2020.

CNRM. HPC Bura. https://cnrm.uniri.hr/bura/, 2021. Accessed: 01-January-2022.

M. A. Coletti, E. O. Scott, i J. K. Bassett. Library for evolutionary algorithms in python (LEAP). In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 1571–1579, 2020.

J. Cona. Developing a genetic programming system. *AI Expert*, pages 20–29, 1995.

C. Darwin. The origin of species by means of natural selection, 1859.

K. Deb, A. Pratap, S. Agarwal, i T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

F. Deserno. Basic Optimization Strategies for CFD-Codes. *Regionales Rechenzentrum Erlangen - RRZE*, 2003.

F. Dierich, K. Wittig, A. Richter, i P. Nikrityuk. Parallelization of the 3d sip algorithm. In *AIP Conference Proceedings*, volume 1648, page 030035. AIP Publishing LLC, 2015.

K. Dolag, M. Reinecke, C. Gheller, i S. Imboden. Splotch: visualizing cosmological simulations. *New Journal of Physics*, 10(12):125006, Dec. 2008. doi: 10.1088/1367-2630/10/12/125006.

J. J. Dongarra i A. J. van der Steen. High-performance computing systems: Status and outlook. *Acta Numerica*, 21: 379–474, 2012.

M. Fenton, J. McDermott, D. Fagan, S. Forstenlechner, E. Hemberg, i M. O'Neill. Ponyge2: Grammatical evolution in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1194–1201, 2017.

F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, i C. Gagné. DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175, 2012.

C. Gagné i M. Parizeau. Open BEAGLE: A New Versatile C++ Framework for Evolutionary Computation. In *GECCO Late Breaking Papers*, pages 161–168. Citeseer, 2002.

C. Gagné i M. Parizeau. Genericity in evolutionary computation software tools: Principles and case-study. *International Journal on Artificial Intelligence Tools*, 15(02):173–194, 2006.

F. Gebali. *Algorithms and parallel computing*. John Wiley & Sons, 2011.

L. Halada i M. Lucká. A Parallel Strongly Implicit Algorithm for Solving of Diffusion Equations. In P. Zinterhof, M. Vajteršic, i A. Uhl, editors, *Parallel Computation*, pages 78–84, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-49164-4.

N. Hansen. The CMA evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

N. Hansen i A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

P. Higuera. olaflow: Cfd for waves. https://olaflow.github.io/, 2017. Accessed: 01-November-2022.

J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1975.

IBM. IBM Spectrum LSF. https://www.ibm.com/docs/en/spectrum-lsf/10.1.0, 2022. Accessed: 01-January-2022.

P. Jamieson, R. Ferreira, i J. A. M. Nacif. GA-lapagos, an open-source c framework including a python-based system for data analysis. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 1589–1590, 2020.

Z. Jin, M. Krokos, M. Rivi, C. Gheller, K. Dolag, i M. Reinecke. High-performance astrophysical visualization using Splotch. *arXiv e-prints*, art. arXiv:1004.1302, Apr. 2010. doi: 10.48550/arXiv.1004.1302.

M. Keijzer, J. J. Merelo, G. Romero, i M. Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 231–242. Springer, 2001.

M. J. Keith i M. C. Martin. Genetic programming in C++: Implementation issues. *Advances in genetic programming*, 1: 285–310, 1994.

J. R. Koza i R. Poli. Genetic programming. In *Search methodologies*, pages 127–164. Springer, 2005.

N. Krasnogor i J. Smith. MAFRA: A Java memetic algorithms framework, 2000.

T. Kruger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, i E. M. Viggen. *The Lattice Boltzmann Method : Principles and Practice*. Springer International Publishing, 2017. ISBN 978-3-319-83103-9.

H. J. Leister i M. Perić. Vectorized Strongly Implicit Solving Procedure for a Seven-Diagonal Coefficient Matrix. *International Journal of Numerical Methods for Heat & Fluid Flow*, 4(2):159–172, 1994. doi: 10.1108/EUM0000000004106. URL https://doi.org/10.1108/EUM0000000004106.

D. Levine. Users guide to the PGAPack parallel genetic algorithm library. *Argonne National Laboratory*, 9700(S 8703941), 1996.

S. López Castaño, A. Petronio, G. Petris, i V. Armenio. Assessment of solution algorithms for les of turbulent flows using openfoam. *Fluids*, 4(3):171, 2019.

S. Luke. *Essentials of metaheuristics*, volume 2. Lulu, 2013.

S. Luke. ECJ then and now. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 1223–1230, 2017.

S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley, i A. Chircop. Ecj: A java-based evolutionary computation research system. *Downloadable versions and documentation can be found at the following url: http://cs. gmu. edu/eclab/projects/ecj*, 880, 2006.

E. Macagno. Fluid mechanics: experimental study of the effects of the passage of a wave beneath an obstacle. *Proceedings of the Academic des Sciences*, 1953.

M. Manhart. A zonal grid algorithm for dns of turbulent boundary layers. *Computers & Fluids*, 33(3):435–461, 2004.

E. Medvet. Hierarchical grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 249–250, 2017.

E. Medvet, A. Bartoli, A. De Lorenzo, i S. Seriani. 2D-VSR-Sim: a simulation tool for the optimization of 2-D voxel-based soft robots. *SoftwareX*, 12:100573, 2020.

E. Medvet, A. Bartoli, F. Pigozzi, i M. Rochelli. Biodiversity in evolved voxel-based soft robots. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 129–137, 2021.

E. Medvet, G. Nadizar, i L. Manzoni. Jgea: a modular java framework for experimenting with evolutionary computation. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 2009–2018, 2022.

J.-J. Merelo, P. Castillo, A. Mora, A. Esparcia-Alcázar, i V. Rivas-Santos. NodEO, a multi-paradigm distributed evolutionary algorithm platform in JavaScript. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1155–1162, 2014.

J. J. Merelo, P. A. Castillo, P. García-Sánchez, P. de las Cuevas, i M. García Valdez. NodIO: A Framework and Architecture for Pool-based Evolutionary Computation. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1323–1330, 2016.

T. Misaka, F. Holzäpfel, I. Hennemann, T. Gerz, M. Manhart, i F. Schwertfirm. Vortex bursting and tracer transport of a counter-rotating vortex pair. *Phys. Fluids*, 24(2):025104, feb 2012. ISSN 1070-6631. doi: 10.1063/1.3684990. URL http://aip.scitation.org/doi/10.1063/1.3684990.

A. S. Monin i A. M. Yaglom. *Statistical Fluid Mechanics*. MIT Press, 1975.

A. Moosaie i M. Manhart. Direct monte carlo simulation of turbulent drag reduction by rigid fibers in a channel flow. *Acta Mechanica*, 224(10):2385–2413, 2013.

J.-B. Mouret i J. Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

H. Mühlenbein, M. Schomisch, i J. Born. The parallel genetic algorithm as function optimizer. *Parallel computing*, 17 (6-7):619–632, 1991.

G. Nadizar, E. Medvet, F. A. Pellegrino, M. Zullich, i S. Nichele. On the effects of pruning on evolved neural controllers for soft robots. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1744–1752, 2021.

F. Nielsen. *Introduction to HPC with MPI for Data Science*. Springer, 2016.

R. Nieto-Fuentes i C. Segura. GP-DMD: a genetic programming variant with dynamic management of diversity. *Genetic Programming and Evolvable Machines*, pages 1–26, 2022.

S. Nolfi. *Behavioral and cognitive robotics: an adaptive perspective*. Stefano Nolfi, 2021.

M. Nowostawski i R. Poli. Parallel genetic algorithm taxonomy. In *1999 Third International Conference on Knowledge-Based Intelligent Information Engineering Systems. Proceedings (Cat. No. 99TH8410)*, pages 88–92. Ieee, 1999.

M. O'Neill i C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

OpenPBS. OpenPBS Open Source Project. https://www.openpbs.org/, 2020. Accessed: 01-January-2022.

E. Pantridge i L. Spector. PyshGP: PushGP in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1255–1262, 2017.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

N. Peller. *Numerische Simulation turbulenter Str{\"o}mungen mit Immersed Boundaries Doktor-Ingenieurs*. PhD thesis, Technische Universit{\"a}t M{\"u}nchen, 2010.

N. Peller, A. L. Duc, F. Tremblay, i M. Manhart. High-order stable interpolations for immersed boundary methods. *International Journal for Numerical Methods in Fluids*, 52(11):1175–1193, 2006.

S. B. Pope. *Turbulent flows*. Cambridge university press, 2000.

A. Rak, L. Grbcic, A. Sikirica, i L. Kranjcevic. Experimental and LBM analysis of medium-Reynolds number fluid flow around NACA0012 airfoil. *International Journal of Numerical Methods for Heat and Fluid Flow*, ahead-of-print (ahead-of-print), 2023. ISSN 09615539. doi: 10.1108/HFF-06-2022-0389/FULL/XML.

A. Ramírez, J. R. Romero, i S. Ventura. An extensible JCLEC-based solution for the implementation of multi-objective evolutionary algorithms. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1085–1092, 2015.

J. S. Reeve, A. Scurr, i J. H. Merlin. Parallel versions of stone's strongly implicit algorithm. *Concurrency and Computation: Practice and Experience*, 13(12):1049–1062, 2001.

P. Renc, P. Orzechowski, A. Byrski, J. Wăs, i J. H. Moore. EBIC. JL: an efficient implementation of evolutionary biclustering algorithm in Julia. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1540–1548, 2021.

M. Rivi, C. Gheller, T. Dykes, M. Krokos, i K. Dolag. GPU accelerated particle visualization with Splotch. *Astronomy and Computing*, 5:9–18, July 2014. doi: 10.1016/j.ascom.2014.03.001.

M. Roser, H. Ritchie, i E. Mathieu. Technological change. *Our World in Data*, 2022. https://ourworldindata.org/technological-change.

A. Rummler. Evolvica: a Java framework for evolutionary algorithms, 2007.

A. Rummler i G. Scarbata. eaLib—A Java Frameword for Implementation of Evolutionary Algorithms. In *International Conference on Computational Intelligence*, pages 92–102. Springer, 2001.

P. Ruol, L. Martinelli, i P. Pezzutto. Formula to predict transmission for π-type floating breakwaters. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 139(1):1–8, 2013.

Y. Sakai i M. Manhart. Consistent flow structure evolution in accelerating flow through hexagonal sphere pack. *Flow, Turbulence and Combustion*, 105(2):581–606, 2020.

Y. Sakai i M. Manhart. Simd-optimisation of the cfd software package mglet for supermuc-ng. https://www.konwihr.de/konwihr-projects/simd-optimisation-of-the-cfd-software-package-mglet-for-supermuc-ng/, 2021. Accessed: 2022-04-29.

Y. Sakai, S. Mendez, H. Strandenes, M. Ohlerich, I. Pasichnyk, M. Allalen, i M. Manhart. Performance optimisation of the parallel cfd code mglet across different hpc platforms. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–13, 2019.

T. Salimans, J. Ho, X. Chen, S. Sidor, i I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

W. Schanderl i M. Manhart. Reliability of wall shear stress estimations of the flow around a wall-mounted cylinder. *Computers & Fluids*, 128:16–29, 2016.

W. Schanderl, U. Jenssen, i M. Manhart. Near-wall stress balance in front of a wall-mounted cylinder. *Flow, Turbulence and Combustion*, 99(3):665–684, 2017a.

W. Schanderl, U. Jenssen, C. Strobl, i M. Manhart. The structure and budget of turbulent kinetic energy in front of a wall-mounted cylinder. *Journal of Fluid Mechanics*, 827:285–321, 2017b.

SchedMD. Slurm Workload Manager. https://slurm.schedmd.com/, 2021. Accessed: 01-January-2022.

E. O. Scott i S. Luke. ECJ at 20: toward a general metaheuristics toolkit. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 1391–1398, 2019.

H. J. Siegel. A model of simd machines and a comparison of various interconnection networks. *IEEE Transactions on Computers*, 28(12):907–917, 1979.

R. E. Smith. A historical overview of computer architecture. *IEEE Annals of the History of Computing*, 10(04):277–303, 1988.

V. Springel, R. Pakmor, O. Zier, i M. Reinecke. Simulating cosmic structure formation with the gadget-4 code. *Monthly Notices of the Royal Astronomical Society*, 506(2):2871–2949, 2021.

A. Stephan, F. Holzäpfel, i T. Misaka. Hybrid simulation of wake-vortex evolution during landing on flat terrain. *International Journal of Heat and Fluid Flow*, 49:18–27, 2014.

A. Stephan, J. Schrall, i F. Holzäpfel. Numerical Optimization of Plate-Line Design for Enhanced Wake-Vortex Decay. *J. Aircr.*, 54(3):995–1010, may 2017. ISSN 0021-8669. doi: 10.2514/1.C033973. URL https://arc.aiaa.org/doi/10.2514/1.C033973.

H. L. Stone. Iterative solution of implicit approximations of multidimensional partial differential equations. *SIAM Journal on Numerical Analysis*, 5(3):530–558, 1968.

R. Storn i K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

H. Strandenes, B. Pettersen, H. I. Andersson, i M. Manhart. Influence of spanwise no-slip boundary conditions on the flow around a cylinder. *Computers & Fluids*, 156:48–57, 2017.

M. Stürmer. Optimierung des Red-Black-Gauss-Seidel-Verfahrens auf ausgewählten x86-Prozessoren. *Studienarbeit*, 08 2005.

Y. Tang, Y. Tian, i D. Ha. EvoJAX: Hardware-Accelerated Neuroevolution. *arXiv preprint arXiv:2202.05008*, 2022.

The HDF Group. Hierarchical data format, version 5. http://www.hdfgroup.org/HDF5/, 1997-2020. Accessed: 01-November-2022.

TOP500.org. TOP500 lists. https://www.top500.org/, 2022. Accessed: 01-November-2022.

A. Törn i A. Žilinskas. *Global optimization*. Springer, 1989.

R. Trobec, B. Slivnik, P. Bulic, i B. Robic. *Introduction to parallel computing: from algorithms to programming on state-of-the-art platforms*. Springer, 2020.

L. Unglehrt i M. Manhart. Onset of nonlinearity in oscillatory flow through a hexagonal sphere pack. *J. Fluid Mech.*, 944:A30, 2022. doi: 10.1017/jfm.2022.496.

L. Vanneschi, M. Castelli, i L. Manzoni. The k landscapes: a tunably difficult benchmark for genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1467–1474, 2011.

S. Ventura, C. Romero, A. Zafra, J. A. Delgado, i C. Hervás. JCLEC: a Java framework for evolutionary computation. *Soft computing*, 12(4):381–392, 2008.

R. Verzicco i R. Camussi. Numerical experiments on strongly turbulent thermal convection in a slender cylindrical cell. *Journal of Fluid Mechanics*, 477:19–49, 2003.

T. Weinzierl. *Principles of Parallel Scientific Computing: A First Guide to Numerical Concepts and Programming Methods*. Springer Nature, 2021.

P. A. Whigham. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, 1995.

D. R. White, J. McDermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaśkowski, U.-M. O'Reilly, i S. Luke. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, 2013.

F. Wilhelmstötter. Jenetics: Java genetic algorithm library. *Abgerufen am*, 4, 2019.

J. H. Williamson. Low-storage runge-kutta schemes. *Journal of Computational Physics*, 35(1):48–56, 1980.

X.-s. Zhang, S. Ma, i W.-y. Duan. A new l type floating breakwater derived from vortex dissipation simulation. *Ocean Engineering*, 164:455–464, 2018.

T. Zhu i M. Manhart. Oscillatory darcy flow in porous media. *Transport in Porous Media*, 111(2):521–539, 2016.

T. Zhu, C. Waluga, B. Wohlmuth, i M. Manhart. A study of the time constant in unsteady porous media flow using direct numerical simulation. *Transport in Porous Media*, 104(1):161–179, 2014.

D. Zongker, B. Punch, i B. Rand. lil-gp 1.0 User's Manual. *Dept. of Computer Science, Michigan State University*, 1, 1995.

Autorship

This manual is published under the following licenses:

Images used for chapter headers

- Cover image: Free to use under the Freepik License @ https://www.freepik.com/ (autor: kjpargeter)
- Contents image: Free to use under the Unsplash License @ https://unsplash.com/ (autor: Aaron Burden)
- Chapter Parallel programming image: Free to use under the Freepik License @ https://www.freepik.com/ (autor: pikisuperstar)
- Chapter References image: CC0 Creative Commons Zero (CC0) @ https://pixabay.com/en/book-open-pages-library-books-408302/