Application of Parallel Computing

Graduate University Study of Mechanical Engineering

Graduate University Study of Computing

Primjena paralelnog računanja

Diplomski sveučilišni studij strojarstva

Diplomski sveučilišni studij računarstva

Faculty of Engineering Rijeka

Center for Advanced Computing and Modelling

November 5, 2020

Contents

1 Introduction **3**

2 Supercomputing Resources at CNRM **4**

3 Basic Terminology **5**

4 Parallel Computer Architectures **6**

4.1 SISD 6

4.2 SIMD 7

4.2.1 SIMD computers with shared memory, vector computers 7

4.2.2 SIMD computers with distributed memory 8

4.2.3 GPGPU processors 9

4.3 MIMD 11

4.3.1 SM-MIMD 11

4.3.2 DM-MIMD 12

4.4 Clusters, Constellations and Grid concepts 13

4.4.1 Cluster 16

4.4.2 ccNUMA 17

4.4.3 Constellation 18

4.4.4 Grid 18

4.5 Moore's Law 19

5 Network Architectures **21**

5.1 Fully Connected Network Topology 22

5.2 Fully Connected Crossbar Network 23

5.3 Multistage Interconnection Network 23

5.4 Hypercube 24

5.5 Fat Tree 25

5.6 2D and 3D Mesh 25

5.7 Bus and Ring topologies 26

5.8 Concluding Remarks 26

6 Current Trends **27**

7 Parallel Programming **28**

7.1 The Dam Break Problem 28

7.2 Job and Data Distribution 29

7.2.1 Job distribution 29

7.2.2 Data Distribution 30

7.2.3 Domain Decomposition 30
7.2.4 Functional Decomposition 34
7.3 Parallel Program Analysis 35
7.4 Amdahl's Law 36
7.5 Gustafson's Law 37
7.6 Reduced Efficiency and its Causes 38
7.7 Parallel Programming Models 39
7.7.1 Parallel Programming on a SM System 39
7.7.2 Parallel Programming on a DM System 41
7.7.3 Parallel Programming Using the Data Distribution Strategy 43

1 Introduction

Parallelism is nowadays evident in majority of the devices that surround us; computers, phones and a wide range of IoT device utilise some type of a 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 utilise sequential code. Serial computing concept (computing on a single processor/using a single process) is a limiting factor since complex problems require immense amount of time. Consequently, parallel computing was born. In essence, parallel computing incorporates hardware components and software into a system which allow seamless simultaneous execution of calculations/code.

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 utilise 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.

2 Supercomputing Resources at CNRM

The Center for Advanced Computing and Modeling is a research and development center, part of the University of Rijeka. The Center collaborates with numerous domestic and foreign scientific institutions and economic entities providing them with the resources and support in order to realize complex tasks in HPC environment. CNRM was built around HPC Bura, a heterogeneous supercomputer owned by the University of Rijeka. Although predominantly used by scientists, Bura is available to all potential users.

Computing resources at CNRM can be divided int three distinct parts:

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

Cluster 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 processor cores are available to users. Each node has 64 GB of memory and 320 GB of disk space, respectively, while the computer nodes together have 18 TB of memory and 95 TB of disk space.

SMP 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, 12 TB of memory and 245 TB of local storage. Two nodes are available for a total of 512 cores.

GPGPU section is a heterogeneous system of four nodes with two Xeon E5 processors (16 cores per node) and two NVIDIA Tesla K40 general purpose GPUs available per node.

HPC Bura can be accessed through two secured nodes built on the Xeon E5 architecture, which are also used to compile and install the software. The supercomputer is powered by Red Hat Enterprise Linux 7 and Slurm Workload Manager.

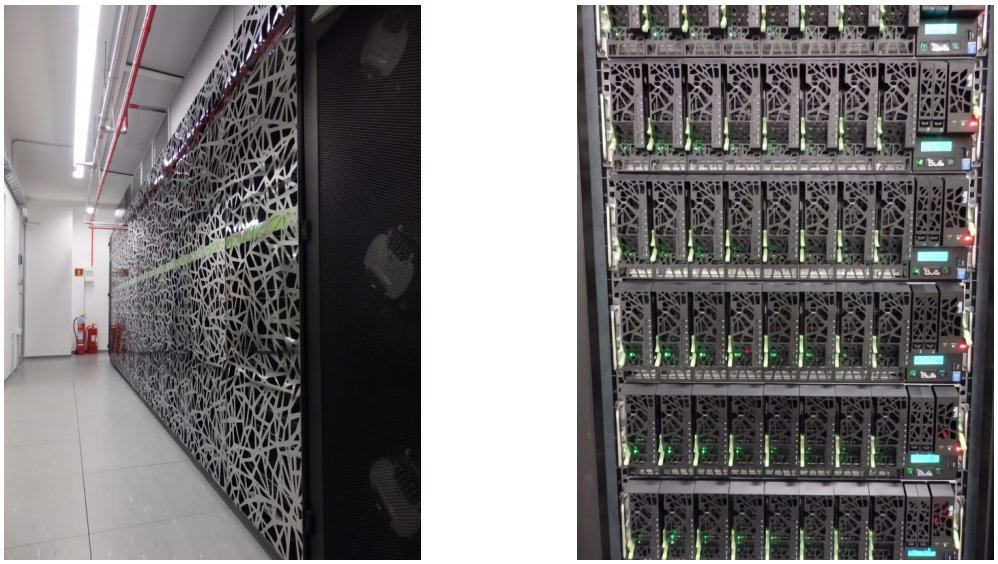

Figure 1: HPC Bura.

3 Basic Terminology

CPU

Electronic circuitry which performs basic arithmetic, logic, controlling and I/O operations specified by the instructions in the program.

Process/Task/Rank

Instance of a computer program that is being executed, a program.

Thread

Code/unit of work that can be scheduled.

Job Scheduling

Assigning tasks to cores. Integral to parallel computing.

Message Passing Interface

Message-passing standard. Allows processes on different cores in distributed memory systems to communicate.

OpenMP

Multithreading implementation that enables parallel programming in shared memory systems.

Flop/s

Floating-point operations per second. Number of floating-point arithmetic calculations systems can perform in a second.

4 Parallel Computer Architectures

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. Programmers 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 an 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 the Flynn's taxonomy. 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:

- **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 own instructions on its local data.

The MISD architecture type will be omitted from further discussions as it is not relevant and has no practical significance in computing. In addition to the previous classifications, parallel computers can be further divided into two principal groups based on memory organization:

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

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. 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, 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 memomry MIMD (DM-MIMD) machine.

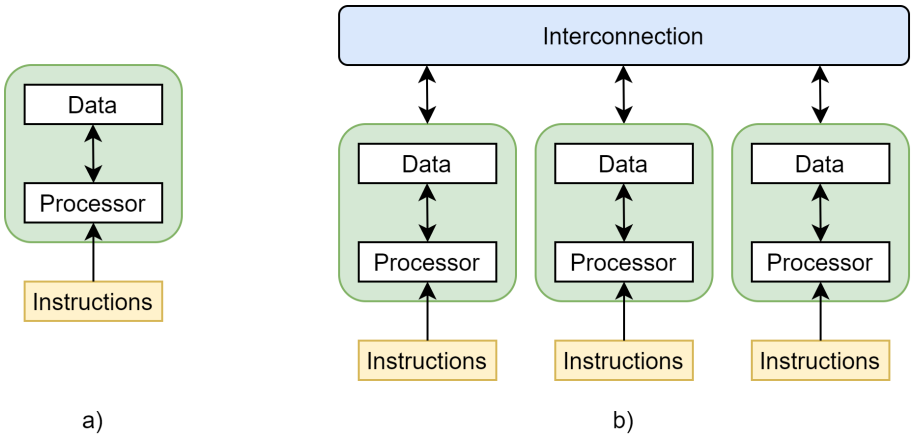

Figure 2: a) SISD computer, b) multicomputer.

4.2 SIMD

Single Instruction Multiple Data (SIMD) computers are a type of a 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. SIMD classification includes two 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 a 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.

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

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

4.2.1 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.

The key distinctiveness of a vector computer is its arithmetic unit, 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 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.

Process of adding two vectors, registers and arithmetic tubes can be seen in figure 3.

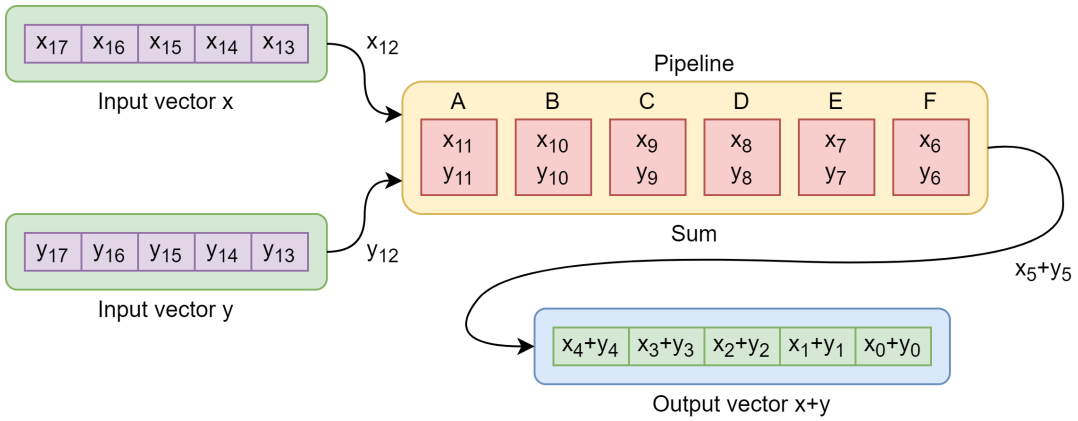

Figure 3: Vector addition on a vector computer.

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

4.2.2 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 utilise 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 machines is always 2D grid. Main parts of a DM-SIMD machine are therefore:

- control processor.
- processor array consisting of many processors (1024 or more) each with its own 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 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 4.

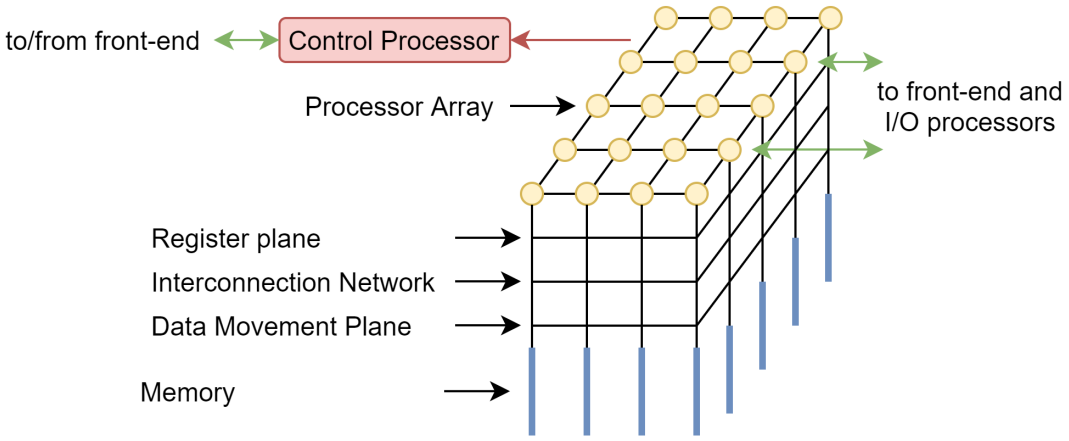

Figure 4: SIMD computer with distributed memory.

4.2.3 GPGPU processors

Majority of the high-performing computers today, both desktop and supercomputers, rely on the 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. This advantage is, however, lost, in cases where considerable communication is needed i.e. communication between different threads.

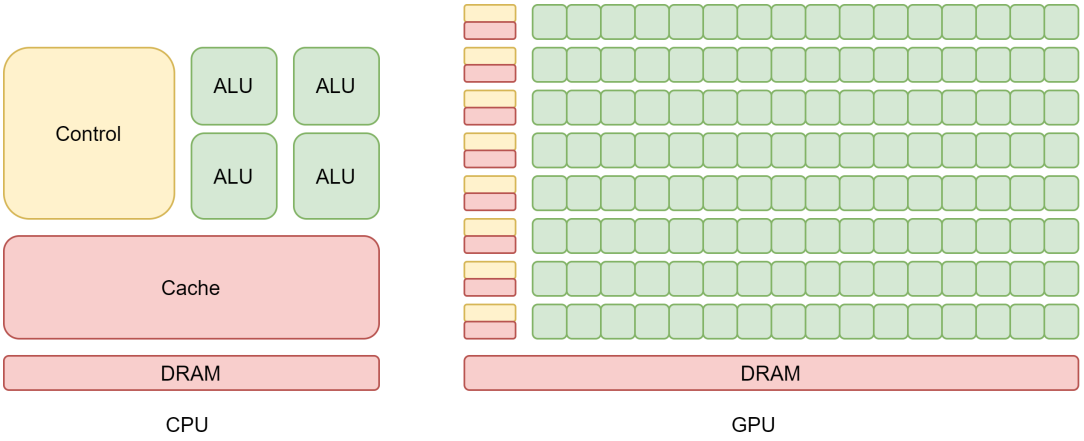

Figure 5: Schematic comparison of CPU and GPU architectures.

Generally speaking, each processor, whether in a CPU or a GPU, has its own cache and access to shared DRAM. GPUs utilise many Arithmetic Logic Units (ALU), when compared to CPU, but lack the shared cache, which limits the overall communication between parallel threads. On the other hand, CPUs, or rather their ALUs, have comparatively slow access to memory i.e. this represents a bottleneck.

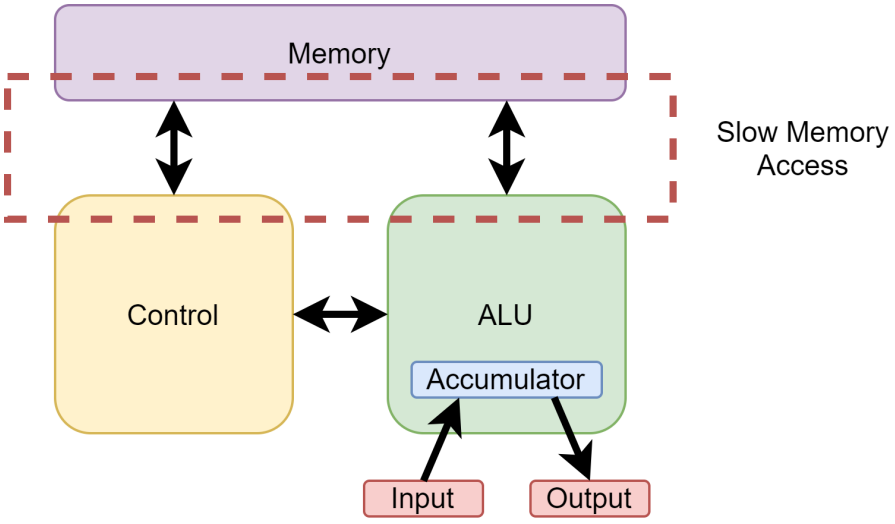

Figure 6: CPU bottleneck: communication pathway between memory and ALU.

Consequently, hybrid designs are usually employed i.e. GPGPU accelerated CPUs/nodes in order to take the advantage of both technologies.

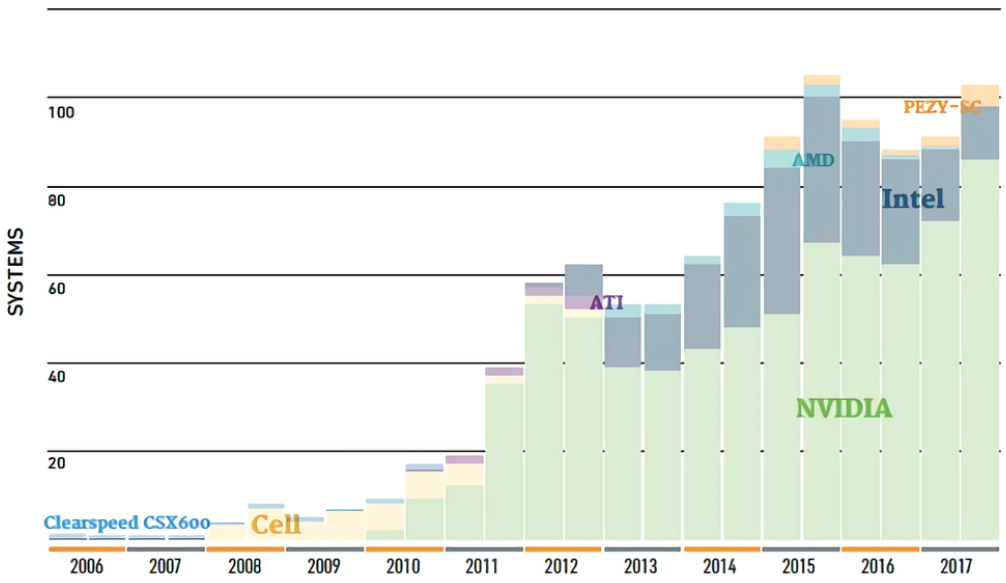

Figure 7: Use of accelerators in supercomputers. GPGPUs are increasingly common, with NVIDIA GPUs dominating the market.

4.3 MIMD

Multiple Instruction Multiple Data (MIMD) group of parallel 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.

4.3.1 SM-MIMD

Processors in shared memory MIMD systems (SM-MIMD) have the ability to 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. The number of processors in a 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 centralized nature of the memory. Architecture of a SM-MIMD computer is given in figure 8.

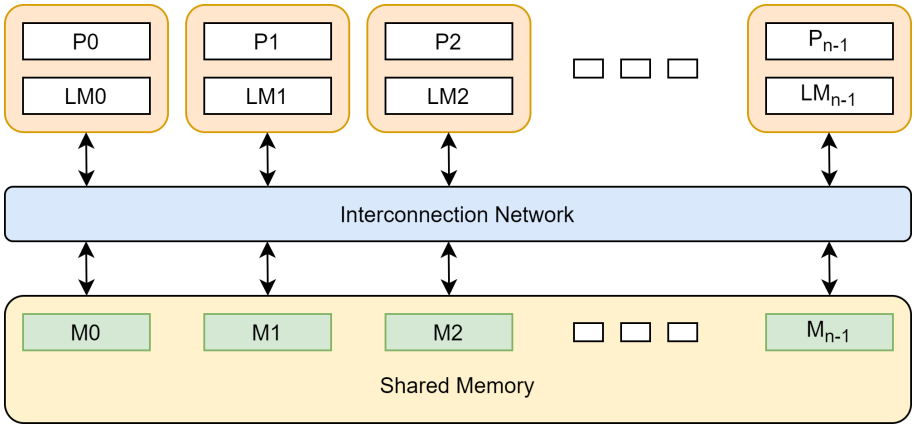

Figure 8: SM-MIMD system.

In addition to standard multiprocessor computers, special types of multiple vector computers can also be considered as 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 vectors processors.

In today's high-performance systems, architecture based on exclusively shared memory concept is rare, with distributed memory systems being a more suitable alternative. This is mostly due to limited scalability of shared memory systems. Primary constraints are:

- organization of memory.
- interconnection networks.
- cache coherency protocols.

ccNUMA (Cache Coherent Non-Uniform Memory Access) architectures are commonly considered a SM-MIMD computers. This is mainly due to the fact that although these computers have physically distributed memory, from a programmers standpoint, they essentially work with shared memory which exists at a logical level (software based).

4.3.2 DM-MIMD

MIMD parallel computers with distributed memory (DM-MIMD) are currently the most prevalent. Unlike MIMD computers with shared memory (SM-MIMD) where the distribution of data is completely transparent to the user, users on a DM-MIMD system must explicitly distribute the data to each processor and explicitly regulate the data exchange between processors. Such requirements from the user/programmer and the overall complexity are the main reasons why this computer architecture was not widely accepted despite being available. 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).
- technological limitations 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.

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, bandwidth scales well with the number of processors. 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 the access to the data that is not stored locally. Consequently, codes are usually more complex.

It is obvious that DM-MIMD parallel machines rely on quality interconnection networks, their topology and bandwidth, which are the main limiting factors in the overall performance efficiency of a DM-MIMD system. Figure 9 depicts a DM-MIMD machine.

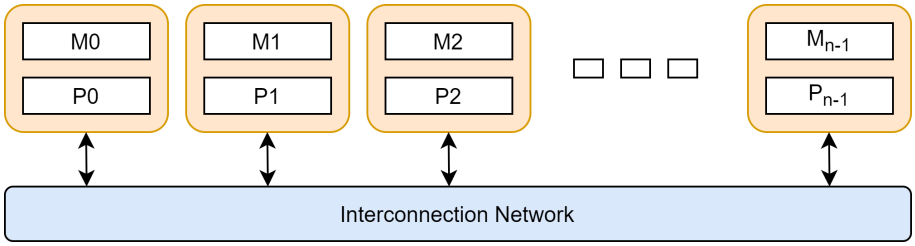

Figure 9: DM-MIMD system.

4.4 Clusters, Constellations and Grid concepts

Clusters and constellations today account for majority of the most powerful supercomputers, as evidenced by the figure 10, which depicts the representation of individual computer architectures among the 500 most powerful supercomputers in the last twenty years. Data is provided by the widely recognized TOP500 supercomputer ranking list. TOP500 list tracks the most powerful computers in the world according to the standard LINPACK test.

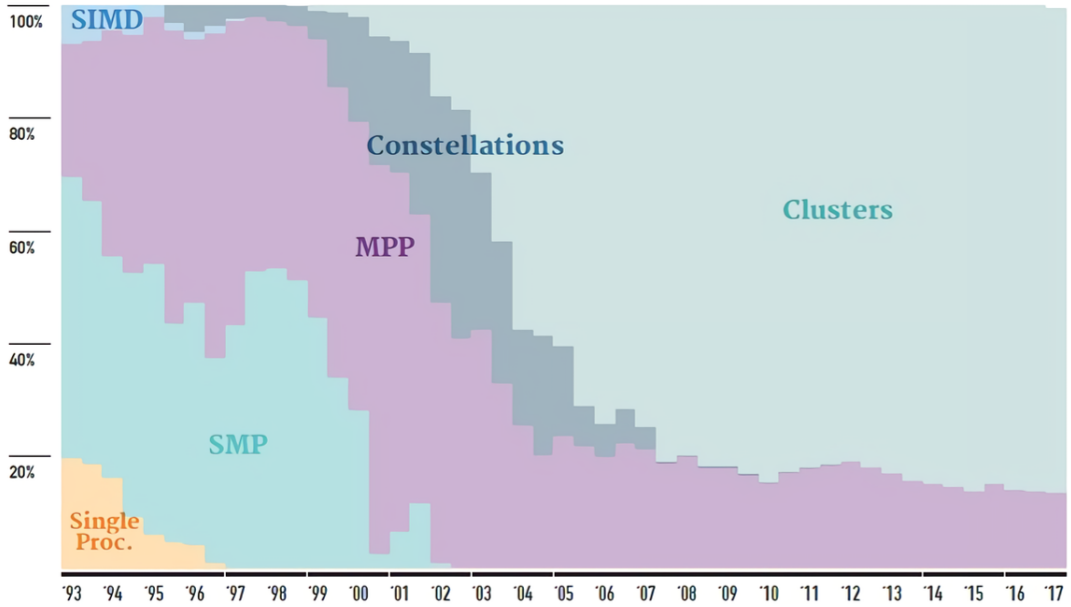

Figure 10: Distribution of major computer architectures in the TOP500 rankings from 1993 to 2017.

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 from a larger volume of RISC processors. SMP's, although often classified as a 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 network and shared a 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. At the beginning of this century, the rise of the clusters began, with an inflexion point being reached in late 2003 when the architecture began to dominate the TOP500 list as shown in figure 11.

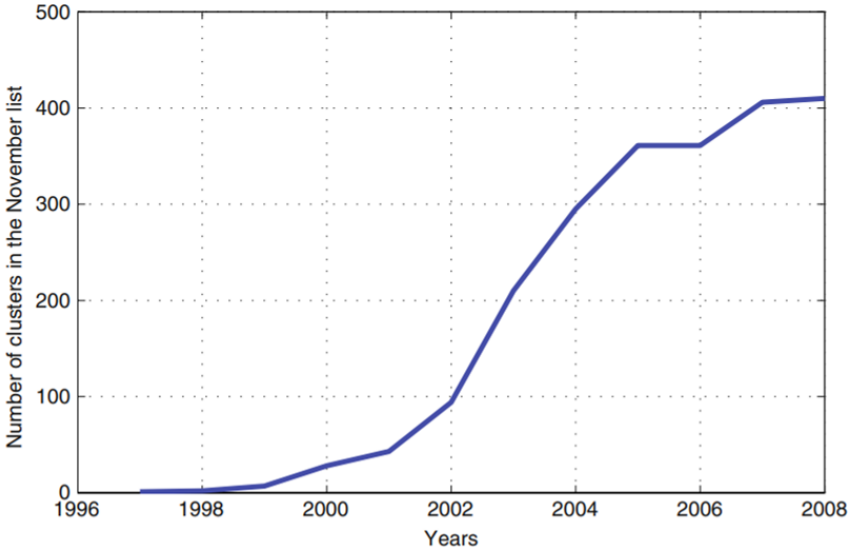

Figure 11: Surge in number of clusters in the years 2002/2003.

Progressive growth of clusters and constellations through the twenty-first century can be attribute 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 utilised in some of the best performing supercomputers. RISC (Sunway, CN) based and ARM (Fugaku, JP) architectures are also present. AMD has only recently managed to revitalize its supercomputer business (Selene, US). Figure 12 show the distribution of different chip architectures from 1993 to 2017.

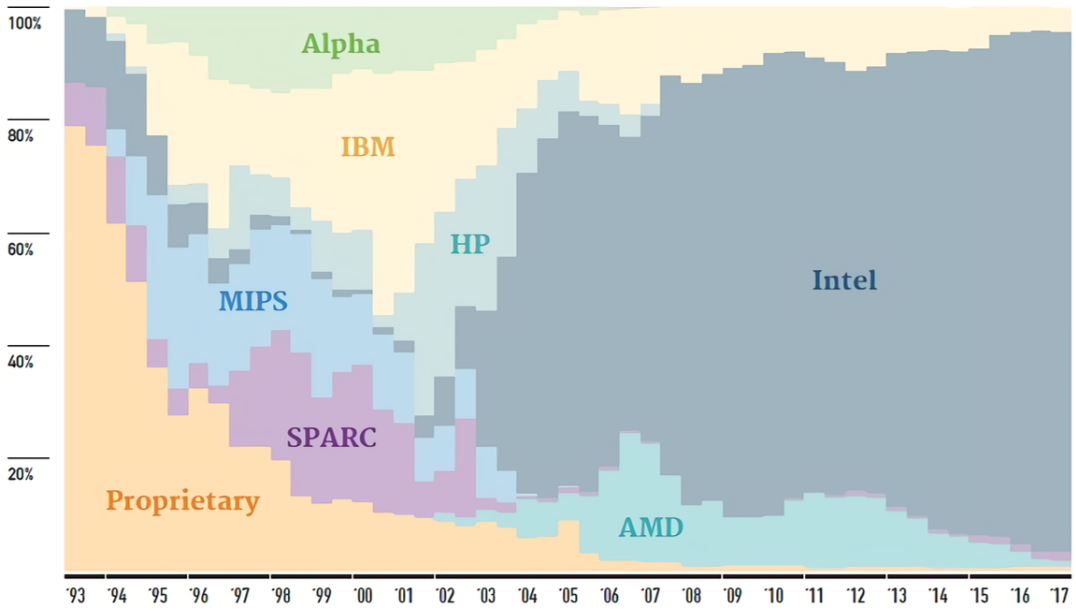

Figure 12: Distribution of chip architectures (technologies) in TOP500 supercomputers.

Current performance leader is Japan's Fugaku supercomputer comprised of ARMv8 based microprocessors and designed by Fujitsu. At it's peak it can almost triple the performance of the #2 supercomputer.

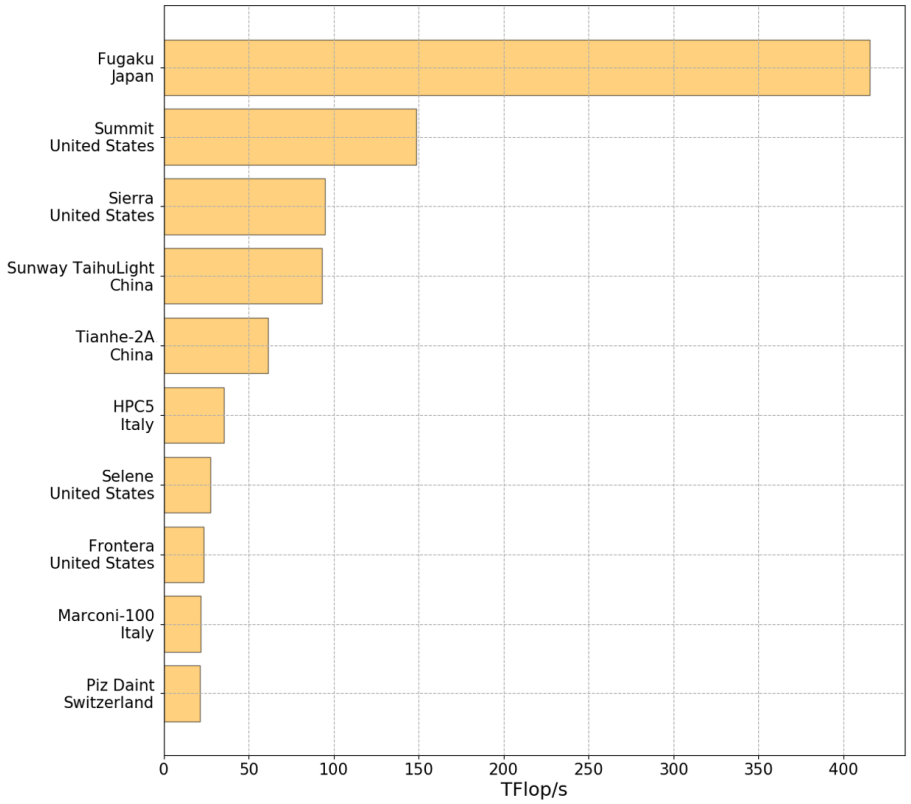

Figure 13: Top performing supercomputers as of June 2020.

4.4.1 Cluster

The concept of cluster architecture experienced a strong surge after the emergence of the first cluster named Beowulf (NASA, 1994). Beowulf clusters are simple computer clusters composed of common PC's connected in a local area network. They are classified as multicomputers as they are built by connecting multiple SISD computers. Beowulf nowadays represents a technology (methodology) of clustering computers to form a parallel, virtual 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). Development of the cluster technology incentivised large computer manufacturers to build their own clusters, hence the Beowulf concept became mostly obsolete and its use limited to scientific computing.

Processing power of cluster computers is nowadays largely based on cheap Intel processors (or alternatives such as AMD/ARM/POWER processors, albeit to less of an extent). Structure of a typical cluster is given in figure 14.

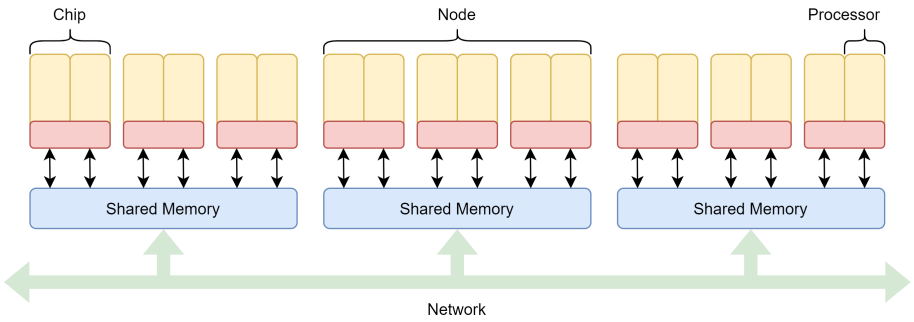

Figure 14: Diagram of an SMP cluster. Compute nodes are comprised of multiple processors connected with a fast interconnect (e.g. crossbar). Multiple nodes are connected with an interconnection network such as hypercube, torus or fat tree.

Displayed sketch defines an architecture in which SMP (Symmetric Multiprocessing) nodes are connected in a cluster through an interconnection network. Processors inside SMP nodes (intranodal) are usually connected with a fast crossbar interconnection which allows 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 or one of the alternatives such as 1GEthernet, 10GEthernet, Infiniband or Myrinet to connect the nodes.

The core of the SMP based cluster designs are massive compute nodes. Compute nodes consist of one or more processors with shared memory. Number of processors per compute node to a larger extent depends on the purpose of the system and the target use (i.e. application use), as well as on the manufacturer and the proposed general concept 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 and called a cluster (in the narrow sense) or as a computer with shared memory (although it utilises physically distributed memory across multiple nodes), which is in essence a ccNUMA (Cache Coherent Non-Uniform Memory Access) concept.

4.4.2 ccNUMA

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 mode. 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 actually physical distributed and shared memory exists only at a logical level, access times can vary significantly, hence the inclusion of the NUMA in the name. Term cache coherency suggests that every variable used 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. There are several ways to 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 analysing 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 a shared memory, ccNUMA systems can be considered SM-MIMD computers, i.e. MIMD computers with shared memory. However, since ccNUMA computers utilise 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.

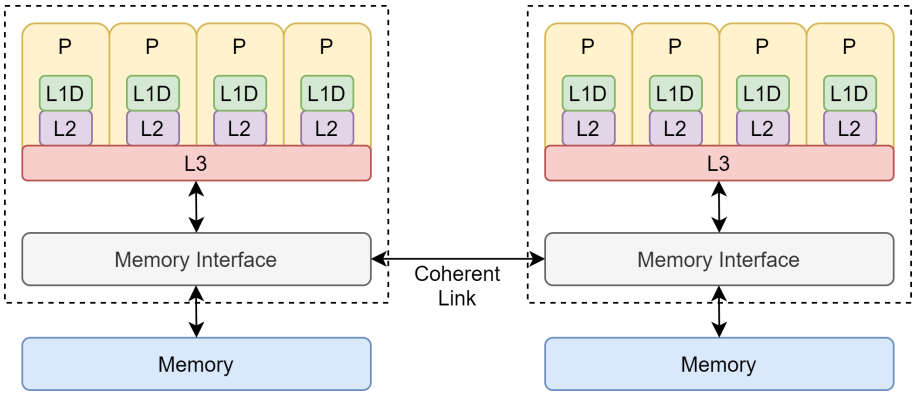

Figure 15: ccNUMA system.

4.4.3 Constellation

Term constellation (in the narrow sense) refers to a specific type of a cluster computer (SMP cluster) in which the number of processors in the compute nodes is greater than the total number of nodes, i.e. systems with massive nodes or nodes in which there is a larger number of processors (at least 4) are often called constellations.

4.4.4 Grid

A grid is a type of a computer system that enables the dynamic use of geographically distributed autonomous subsystems depending on their availability, capacity, performance and price. Grid computer are used to calculate large-scale computational problems such as N-body simulations, seismic simulations, atmospheric and oceanic simulations or protein folding (e.g. foldingathome.org). At its core, a grid computer is a cluster in which the LAN is replaced with a WAN.

4.5 Moore's Law

The number of components (i.e. transistors, resistors, diodes, or capacitors) in an integrated circuits doubles every two years (revised in 1975.). 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 16 depicts the increase in the transistor count since the 70s.

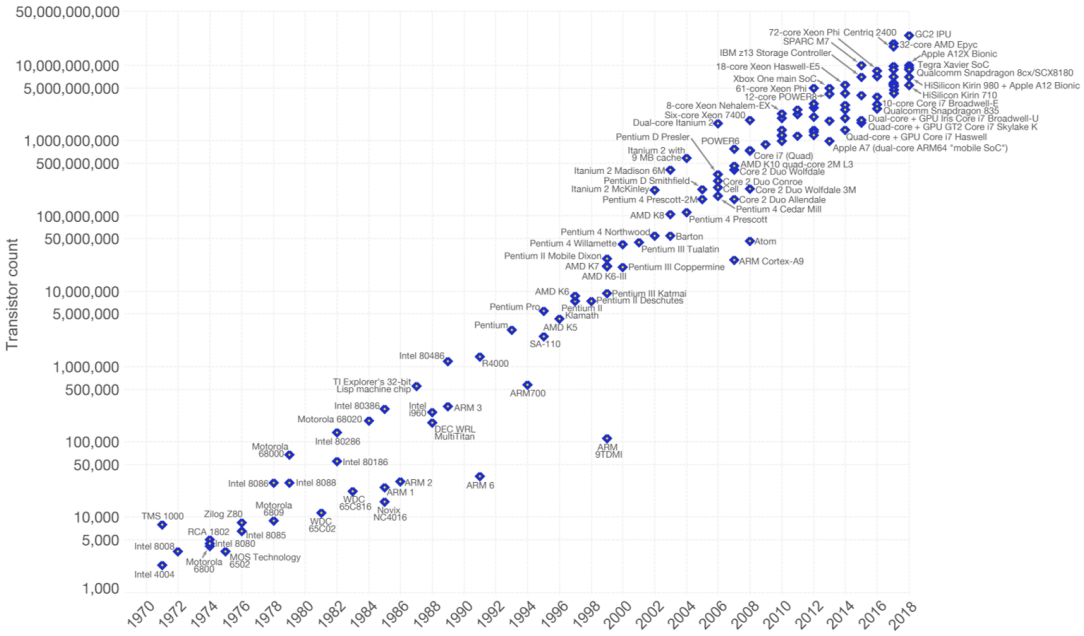

Figure 16: Increase in transistor count since the early 70s.

Moore's Law has held true for almost 50 years, albeit somewhat slowed down as of 2016/2017 with estimates that say the time span has increased from two years to approximately two and a half years. In other words, certain computer minimization technologies are reaching physical limits of what is possible, hence in order to keep the continuation of the increase in computing power, development of new technologies and approaches is of great importance.

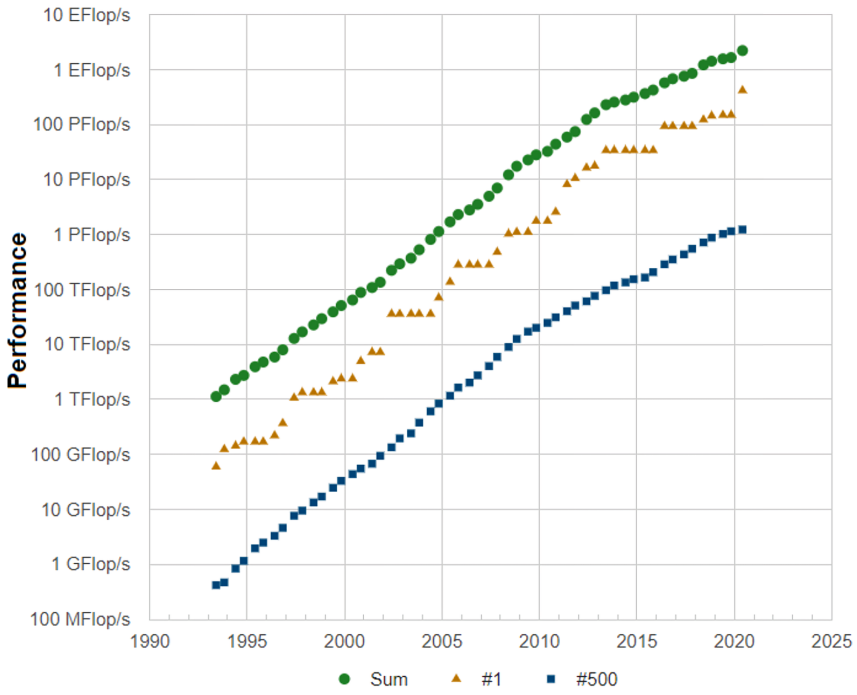

Figure 17: Supercomputer performance development (increase) since the early 90s.

5 Network Architectures

From a computational efficiency standpoint, a network which allows direct connection between each processor in a system, i.e. a fully connected network, is a superior connection design compared to any other option. 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 the topology. Routing is the process of selecting an optimal path for traffic in a network. Flow control is the process of managing the rate of data transmission between nodes whilst the network topology is the arrangement of various elements (e.g. communication nodes and channels) in an interconnection network. Amongst mentioned, network topology is the most troublesome, as the overall efficiency of the network depends on it, yet 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 including latency, bandwidth and scalability. Latency is the time it takes for a message transfer through the network. Bandwidth is the number of bits that can be transmitted in a unit time. Scalability represents the ability of a network to accept a larger throughput while retaining the same performance properties. 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 tasks.

Latency

Latency represents the time that elapses from the moment the message is sent to the moment when the message reaches its destination. 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 receive 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 theoretically maximum data transfer rate through one channel in a given time. 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 time period is called throughput. It is commonly refereed to as 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 classified according to graph theory, where a network is modelled 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 standardization parameters which can be used to define topological properties and performance of the network. These include:

- **path** represents a path from one node to another.
- **link** is a direct link between e.g. two nodes.
- **routing distance** d is the minimum number of connections (links) between two nodes; for fully connected networks distance $d = 1$.
- **diameter** Ω is the maximum routing distance between two nodes in the network.
- **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.

The aforementioned parameters are used to classify and describe some of the most common network topologies presented in upcoming sections.

5.1 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 application).

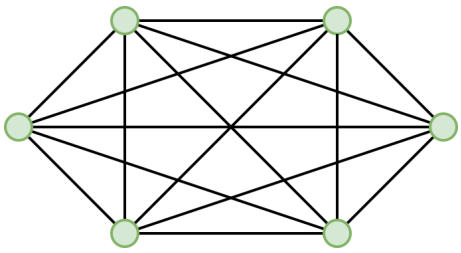

Figure 18: Fully Connected Network Topology.

5.2 Fully Connected Crossbar Network

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 change in the state of switches incorporated in the network. 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 which regulate "queues". For a MIMD system with a large number of processors, e.g. RISC, INTEL or AMD design, this network topology would present a technologically too complex of a design with significant underlying cost. On systems with a relatively small number of nodes, however, especially if those nodes possess significant computing power, e.g. parallel vector machines, 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.

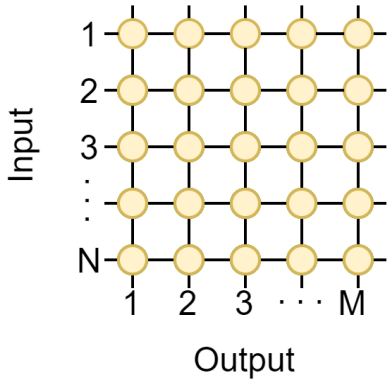

Figure 19: Fully Connected Crossbar Network.

5.3 Multistage Interconnection Network

One of the solutions to the crossbar topology problem are multistage interconnection networks. They too belong to the dynamic subgroup of networks. Inputs and outputs are connected through a stage (set) of switches. These switches are fewer in numbers, 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. 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).

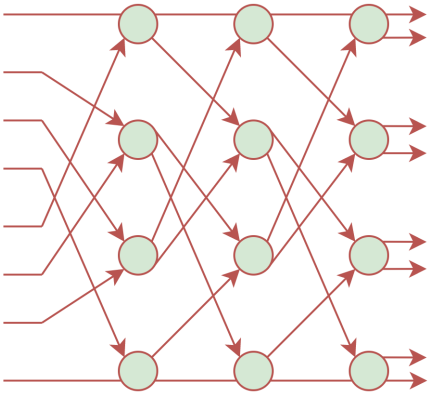

Figure 20: Omega Network.

5.4 Hypercube

In an effort to better balance the relationship between the network quality and price, networks have been developed based on the principle of a so-called hypercube. Each node in a hypercube is connected to n neighbors (diameter equals $n = log_2 N$), therefore the network can connect $N = 2^n$ nodes. Average distance is $n/2$. The dimensionality of the cube is n, hence they are commonly called n-cube networks.

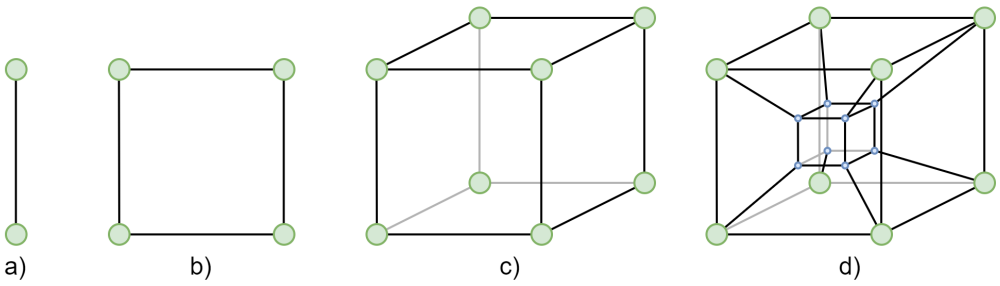

Figure 21: Principle of hypercube network topology, routing distance d and diameter Ω: a) $d = 1, \Omega = 1$, b) $d = 1...2, \Omega = 2$, c) $d = 1...3, \Omega = 3$, d) $d = 1...4, \Omega = 4$.

One of the key benefits is its 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, network would continue to function normally. Two-dimensional meshes and trees can be embedded in a hypercube in such a manner in which 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 resurgence.

5.5 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 simple tree. Congestion for a simple tree typically occurs near the root of the tree due to the concentration of messages that traverse through higher levels of the tree before descending to the target nodes. The fat tree solves this problem by introducing additional connections at those tree levels, thus increasing the bandwidth.

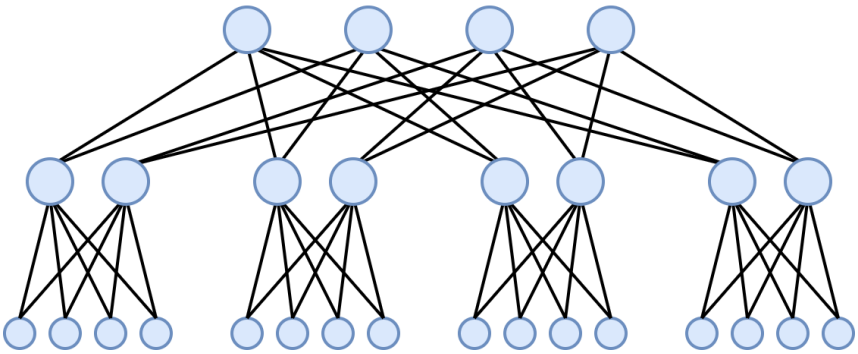

Figure 22: Fat Tree topology.

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

5.6 2D and 3D 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

Processors in 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. If we assume that there are n nodes in the network, parameters of the network are as follows: $d \approx 2\sqrt{n}$, $\Omega \approx 2\sqrt{n}$, complexity $2n$ and connectivity 2.

2D Torus

This design nullifies the topological deficiencies of the 2D mesh by arranging members in a toroidal structure. 2D torus topology allows for the processors on the edges to have the same number of neighbors as a typical centrally located processor in a 2D mesh.

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 neighbouring nodes, which in three dimensions is six. Similar deficiencies on the edges are apparent and are mitigated in 3D torus designs. For n nodes, parameters of the network are: $d = 1...\sqrt{n}$, $\Omega \approx 2\sqrt{n}$, complexity $2n$ and connectivity 4.

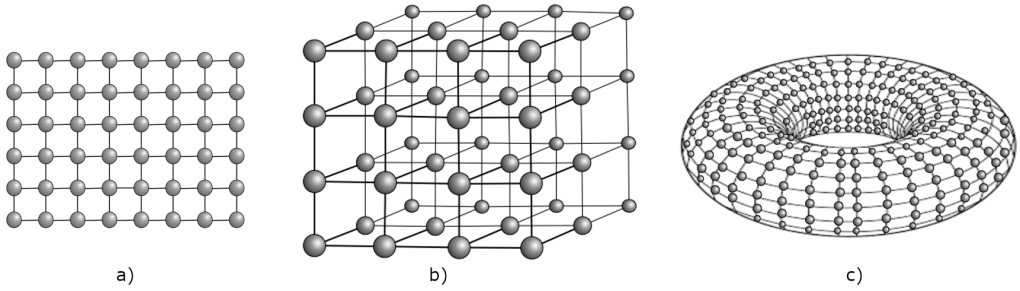

Figure 23: 2D and 3D Meshes: a) 2D Mesh, b) 3D Mesh, c) 2D Torus.

5.7 Bus and Ring topologies

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

The ring is one of the oldest types of networks. Processors in a ring are arranged in a consecutive, linear fashion, with every processor having only two neighbors.

5.8 Concluding Remarks

In general, the most widespread network topologies used in the construction of heavily parallelized machines are based on the following concepts: hypercube, fat-tree, torus and full crossbar.

It is important to point out the increasing relevance of the interconnection networks, which is a direct consequence of the rapid increase in the number of processors per parallel computer.

Due to the immense increase in the number of processors per parallel computer, interconnection networks in recent years are designed to accommodate a median (average) congestion, rather than being built to provide adequate bandwidth to satisfy the so-called worst case.

Current design philosophy entails traffic congestion, which is deemed acceptable due to cost savings. Consequently, nowadays 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 whole field is extremely attractive from an engineering and mathematical standpoint.

6 Current Trends

Even though semiconductor sizes are continuously being shrunk, with each generation physical limits are of increasing concern. 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 performance of massively parallel architectures. Multicore systems (dozens or even hundreds of cores) with simultaneous multithreading and asymmetric execution are of increasing interest. Consumer GPU's as of 2020 have upward of 10000 cores. Existing massive core designs are slowly being replaced by simple power efficient cores (ARM), which, although comparatively worse, offer better value both in terms of initial investment as well as general performance/$.

a) b)

Figure 24: a) Fugaku supercomputer - ARM-based supercomputer with over 7 million cores. b) RTX 3090 - consumer grade GPU with 10496 CUDA cores.

7 Parallel Programming

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, proper programming approach can be adopted, in order to optimally utilise the parallel architecture.

Parallel programming concepts and models will be assessed on an example 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.

7.1 The Dam Break Problem

As previously asserted, parallel programming will be assessed on a dam break test case. Computational domain is comprised of two water tanks separated by a dam. Water level is higher at the left tank. Figure 25 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 $\mathbf{A}(1:n, 1:m)$.

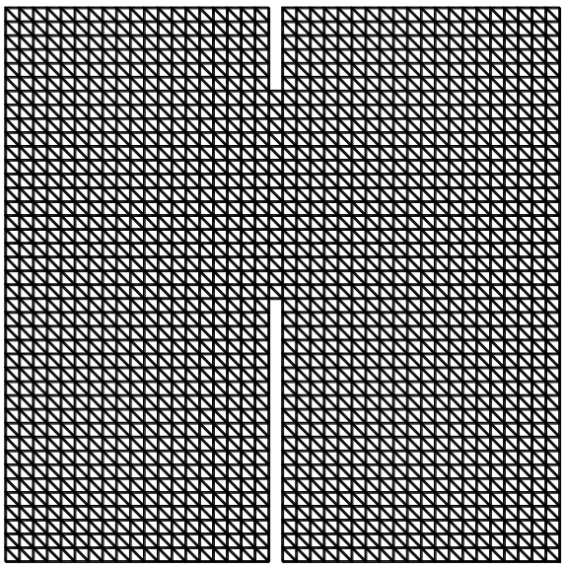

Figure 25: Numerical domain of a dam break test case.

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:

$$(\mathbf{A} + \mathbf{B} + \mathbf{C}) \cdot \mathbf{x} = \mathbf{d} \tag{1}$$

Solution to this problem can be attained with a sequential code such as that given in 1.

Code 1: Dam break sequential code algorithm.

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

7.2 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.

7.2.1 Job distribution

The strategy behind the job distribution when calculating expression 1 is shown in code snippet 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.

Code 2: Dam break job distribution.

```
1  compute A, iterations 1−25     → assign to processor 1
2  compute A, iterations 26−50    → assign to processor 2
3  compute A, iterations 51−75    → assign to processor 3
4  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 of 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, 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.

7.2.2 Data Distribution

Data distribution strategy that would solve the problem outlined in equation 1 is given in snippet 3. Given example of the data distribution among four processors for matrix **A** should be implemented for remaining matrices and vectors as well.

Code 3: Dam break data decomposition.

```
1 A(1:20,1:50)       → data assigned to processor 1
2 A(1:20,51:100)     → data assigned to processor 2
3 A(1:20,101:150)    → data assigned to processor 3
4 A(1:20,151:200     → 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 an MIMD architecture, but can also be used on a SIMD machine.

7.2.3 Domain Decomposition

In engineering practice, the most common parallelization strategy is the domain decomposition strategy shown in figure 26. 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, 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 or at runtime, 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).

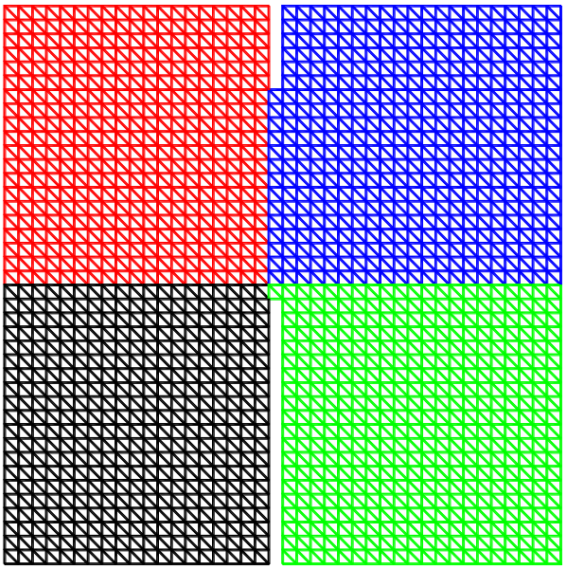

Figure 26: Dam break domain decomposition using four processors.

Figure 27 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 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.

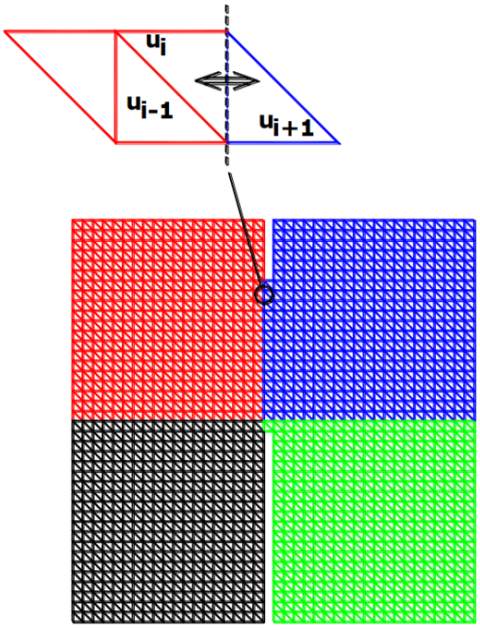

Figure 27: Communication between boundary elements of subdomains.

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, 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 28.

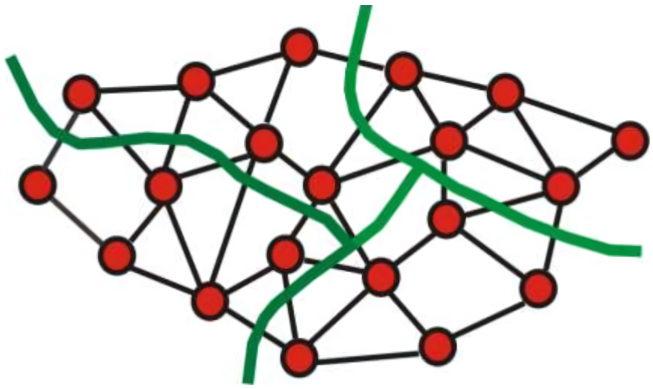

Figure 28: Decomposition using graph theory.

From a computational standpoint, every vertex (point) in figure 28 represents a computational load while the edges correspond to communication. The goal is to minimize the communication (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.

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, random method and scattered scheme can be utilised. 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. 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. 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 being backwards-reconstructed to its original state with the partitioning repeated for every level of refinement. KL method is commonly used to re-partition each intradivision. Main phases of a multilevel method are shown in figure 29.

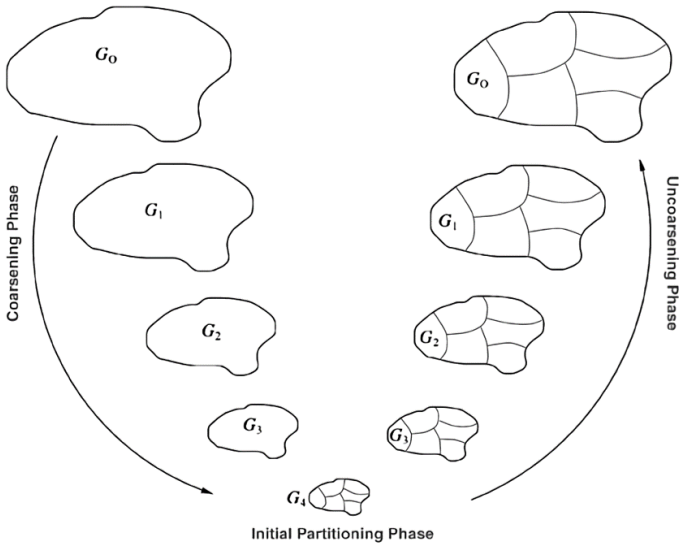

Figure 29: Phases of a multilevel method.

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.

7.2.4 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 aircrafts, 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 Euler approach can be efficiently decomposed.

7.3 Parallel Program Analysis

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 time required to solve the same problem on a single processor. Speedup $S(p, N)$ can consequently be calculated as:

$$S(p, N) = \frac{T(1, N)}{T(p, N)} \tag{2}$$

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:

$$S(p, N) = p \tag{3}$$

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:

$$E(p, N) = \frac{S(p, N)}{p} \tag{4}$$

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 addresable 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, scalability $Sc(p, N)$ coefficient is defined:

$$Sc(p, N) = \frac{N}{n} \tag{5}$$

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

$$T(1, N) = T(p, N) \tag{6}$$

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. 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 which minimizes waiting and synchronization times i.e. processors should be assigned jobs which 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.

7.4 Amdahl's Law

Theoretically, 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 then 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:

$$T(1, N) = s + q \tag{7}$$

For p processors the total time equals:

$$T(p, N) = s + \frac{q}{p} \tag{8}$$

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

$$S(p, N) = \frac{T(1, N)}{T(p, N)} = \frac{s + q}{s + \frac{q}{p}} \tag{9}$$

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

$$S(p, N) = \frac{1}{1 - q + \frac{q}{p}} \tag{10}$$

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:

$$S(p, N) < \frac{1}{1 - q} \tag{11}$$

The asymptotic behavior of the speedup curves is therefore, according to the Amdahl's law, understandable and expected. Speedup curves as functions of the number of processors for different

cases are shown in figure 30. If we consider the case that has been ran 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 minimum, in order to achieve appropriate efficiency and speedups, which is typically the case for large test cases with extensive databases.

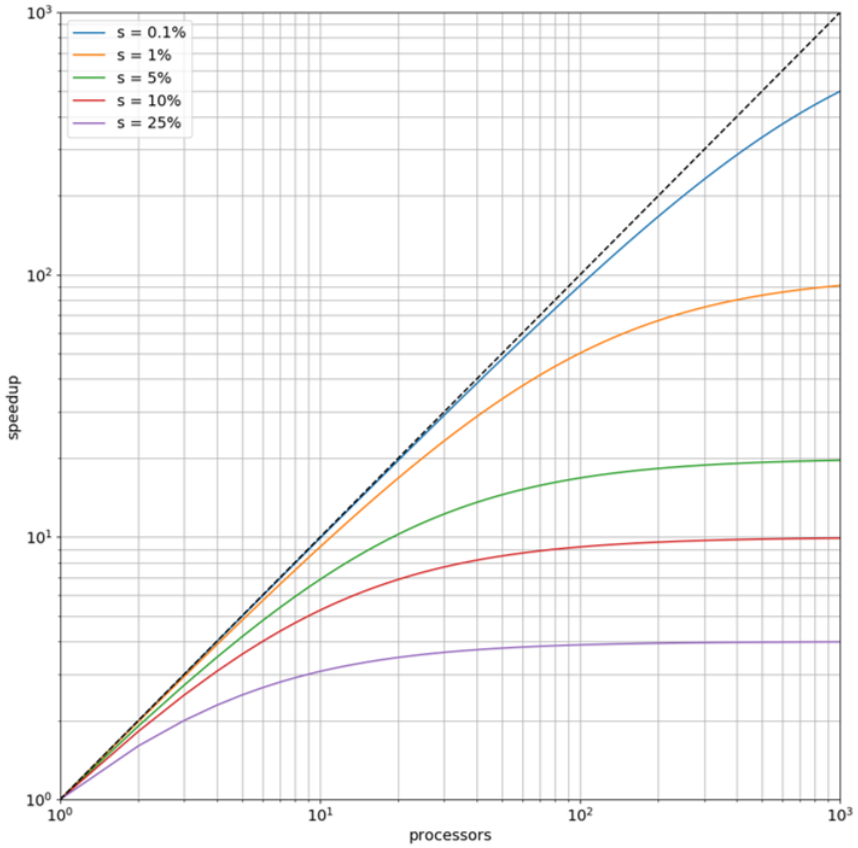

Figure 30: Amdahl's Law.

7.5 Gustafson's Law

One of the key assumptions in the 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 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:

$$T(p, N) = s + q \tag{12}$$

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

$$T(1, N) = s + q \cdot p \tag{13}$$

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

$$S(p, N) = \frac{T(1, N)}{T(p, N)} = \frac{s + q \cdot p}{s + q} \tag{14}$$

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

$$S(p, N) = 1 - q + q \cdot p \tag{15}$$

Speedup curves as functions of the number of processors for different cases according to Gustafson's law are shown in figure 31.

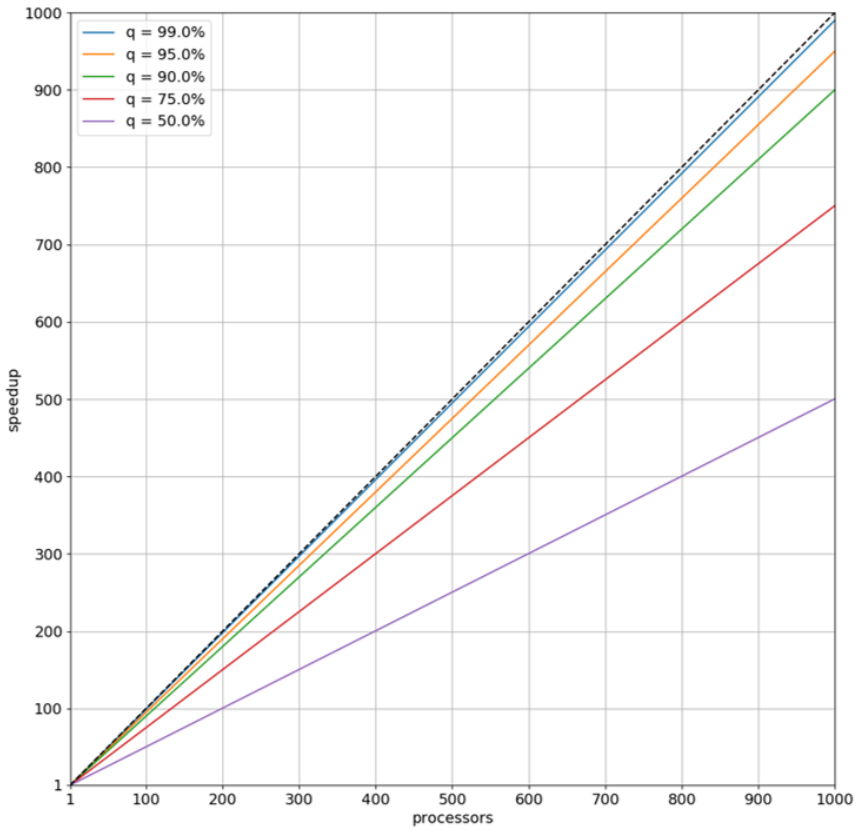

Figure 31: Gustafson's Law.

7.6 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 request, 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 on 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. It is evident that 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.

7.7 Parallel Programming Models

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.

7.7.1 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 (1):

Code 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 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 32).

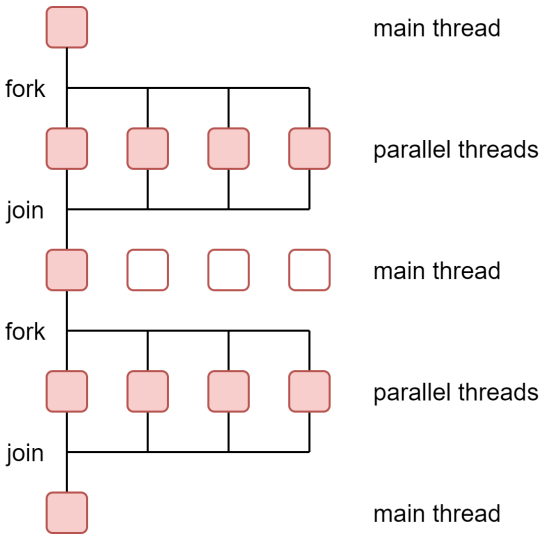

Figure 32: OMP fork-join model.

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.

7.7.2 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 33 shows a master-worker paradigm when four processes are utilised.

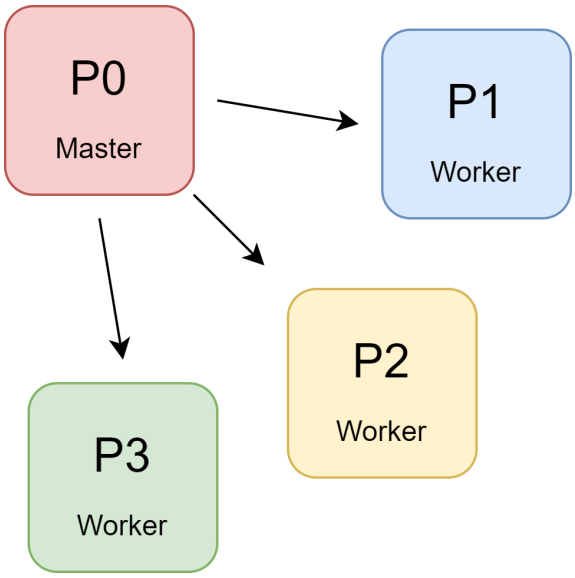

Figure 33: Master process P0 initializes workers P1-P3 and distributes data if needed.

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 34).

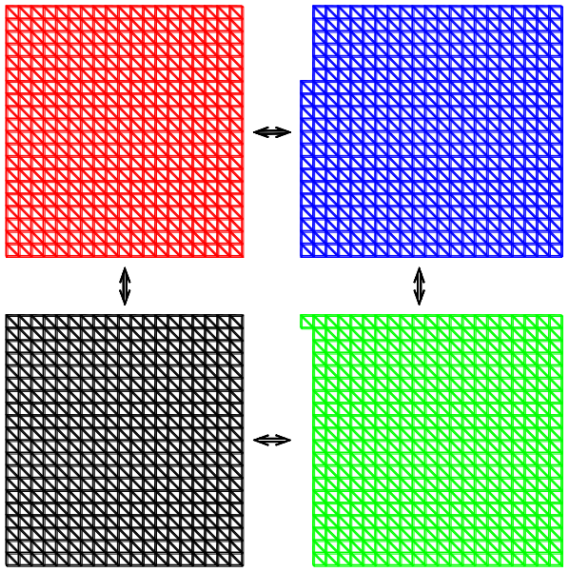

Figure 34: Communication is explicitly defined by the programmer when using MPI standard.

Sequential code depicted in 1 can be parallelized according to the MPI standard:

Code 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(...)
```

7.7.3 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 6 depicts HPF parallelization approach.

Code 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.