Politecnico di Torino

PhD School

High Performance (Scientific) Computing: from your desktop PC to parallel supercomputers A user-friendly introduction

Michele Griffa
PhD student, XIX cycle

Dept. of Physics, Politecnico di Torino
corso Duca degli Abruzzi 24, 10129, Torino (Italy)
and
Bioinformatics and High Performance Computing Laboratory
Bioindustry Park of Canavese
via Ribes 5, 10010, Colleretto Giacosa (Italy)

E-mail: michele.griffa@polito.it
Personal Web site: http://www.calcolodistr.altervista.org

16/10/06

Table of contents

Group picture of the teachers (red numbers) and participants to the 13th Cineca Summer School on Parallel Computing. Instructors: (1) Sigismondo Boschi, (2) Carlo Cavazzoni, (3) Cristiano Calonaci, (4) Claudio Gheller, (5) Gerardo Ballabio. Giovanni Erbacci and Chiara Marchetto (the other teachers) were not present in this photo. Green arrow (and green t-shirt): me. Many thanks to C. Calonaci and Cineca Web Administrators for having recovered this photograph.

Group picture of the teachers (red numbers) and participants to the 2nd Caspur Summer School on Advanced HPC. Instructors: (1) Federico Massaioli, (2) Giorgio Amati, (3) Gabriele Mambrini, (4) Roberto Verzicco, (5) Jack Dongarra. (6) Elena ?????. The other teachers of the School were not present in this photo. Green arrow (but white t-shirt, this time): me.

1. Introduction

This document is a brief and incomplete report on the art of **High Performance** (Scientific) **Computing** (**HPC**) targeted to Scientists (certainly not to Computer Scientists or Computational Scientists). Its main objective is to give the interested non-specialist reader an overview on how to improve the performance of his/her computing codes used in his/her scientific day-work. Performance does not mean only speed of execution. Indeed, high performance is concerned with the trade-off between speed, machine workload, volatile/hard memory occupation.

This report should show that for improving the performance of one's scientific computing code it is not necessary or sufficient to have high performance hardware facilities: algorithms design, methods and approaches in the implementation of the code with a programming language, customization of the code according to hardware features are important issues yet.

Section 2 presents a brief introduction to the world of HPC, with an example of real-world computing problem that for being solved requires computing resources that nowadays are still not available. This is an example of grand-challenge problem and it is useful in showing the importance of HPC in every-days human activities.

Section 3 is completely dedicated to computers architectures with a specific logical flow that starts from the basic ingredients of the Von Neumann architecture (the basic logical model for a sequential electronic computer) and arrives at the main architectures of nowadays parallel supercomputers. The intermediate path covers, in a chronological and logical way, the main important steps that has led from single desktop PCs to parallel architectures, with a particular attention to the hardware technological solutions that have tried to introduce parallelism also within the way of operating of a single CPU. Section 3.1 is dedicated to the Von Neumann architecture of a serial computer with a single processing unit (CPU) and its bottlenecks. Section 3.2 is a temptative and incomplete presentation of the evolution of CPU architectures and technologies that have tried to improve their performances, with a stress on technological solutions to implement parallelism within a single processing unit. This choice of concentrating the attention on such issue is in contrast with usual treatments and presentations within non-specialized mass-media contexts, which tend to emphasize the great and powerful increase of CPU speed in terms of number of clock cycles per unit of time (clock frequency). As it might be shown in Section 3.2, the fantastic improvements in CPU performances are not only based on the increase in CPU speed but also on other technical solutions to exploit parallelism. The famous Moore's law, which has seemed to govern the developments of microelectronics in the last three decades and to be at the basis of every leap forward in computing power, is not all the world ! This fact has been getting very clear to hardware and software manufacturers in the last years: physical limits to device integration at the micron scales have been reached, although nanotechnologies promise to beat new limits. However, parallelism seems nowadays to be the only possible solution to maintain a positive rate for computing power. Section 3.3 is dedicated to multiprocessor computers, with a brief overview of the different ypes of architectures and technological solutions to realize them. This part concludes with a brief presentation of Grid Computing, as a kind of parallel computing solution, its promises and its limits.

Section 4 and 5 are entirely dedicated to HPC with the point of view of a user who wants to obtain the best performance from the facilities he/she has, being them a single commodity PC or a big parallel supercomputer.

Section 4 consists in a collection of treatments and tricks to improve the performance of one's computing codes on a single desktop PC, trying to show that off-the-shelf CPUs have many resources that could be exploited to obtain great results in scientific computing. HPC can just be obtained, firstly, for serial codes running on off-the-shelf desktop PCs: the report considers some aspects of code optimization, using all the resources of the serial PC. In Section 4.2, some general rules are considered in order to improve the performance of the serial code according to general features common to nowadays CPUs architectures available on the desktop market. I treat here only the x86 architectures, also known as IA_32 (32 bit architectures), of modern CPUs. Since 2004, x86_64 (64 bit architectures) CPUs have been available on the high-level desktop market. Nowadays, they are commonly used on all categories of computers, from desktop to mobile, from servers to clusters. 64 bit CPUs compatible with the IA_32 instruction set (firstly introduced by AMD then by Intel, implemented in Pentium IV from the Procona version) improve performance in fixed point and floating point arithmetics and, more important, can address RAM memories larger than 4 GiBytes, which is the main objective they have been designed for. Although, x86_64 CPUs are the standard in the design of nowadays multiprocessor supercomputers, the optimization of code running of them require additional knowledge of hardware features which are beyond the scope of this report. Anyway, some of the considerations presented below for x86 architectures remain valid for x86_64 ones.

In Section 4.3, other types of optimization operations and methods are described, which can be realized not at the source code level but at the Assembly code level, using specific option flags implemented in modern compilers. I consider there and along the whole report the GNU/Linux OS as the standard environment of work and the GCC (GNU Compiler Collection, version 4.1.1) as the standard compiler set therein. This choice is motivated by the fact that GNU/Linux OSs and software are *de facto* the standard tools for Scientific Computing, particularly for cluster computing. Section 4.4 introduces some features of the Intel Fortran-C/C++ compiler, which is released by Intel freely according to a non-commercial license for personal use. This compiler is optimized for Intel CPU architectures and is a good example of a tool for improving the performance of serial codes according to the specific hardware features of the CPU in use.

When scaling issues imply that the resources of a single PC are not sufficient for a real-world execution of the code, it is necessary to implement a parallel version of the code, to be run on multiprocessor computers. This report is concerned with architectures of parallel supercomputers, including clusters of distributed heterogeneous PCs which are often called as resources of *Grid Computing*. In Section 5.1, the two main parallel programming paradigms are considered along with the corresponding most common and used softwares/methods of implementation developed in the last decades.

After some concluding remarks in Section 6 and a brief Bibliography/list of resources in Section 7, I present a collection of useful WWW resources (links) on parallel computing and more general on HPC in Section 8.

This report is based on knowledge and documentations that I have been collecting since 2004, when I attended the 13th edition of the Summer School on Parallel and High Perfomance Computing at Cineca (Consorzio InteruNiversitario per l'Elaborazione e il Calcolo Automatico), placed in Casalecchio di Reno (Bologna). The school is organized every year for University, undergraduates and graduate students and is dedicated to a general introduction into HPC, including parallel computing. The school lasts two weeks (8 hours/day) and for me it was the first occasion for learning about HPC. Part of the material presented below is based on the slides and documents of the school, so I am very grateful to Dr. G. Erbacci, who leads the HPC division of SAP Department of Cineca, and to the other teachers of the school (Dr. G. Ballabio, Dr. S. Boschi, Dr. C. Calonaci, Dr. C. Cavazzoni, Dr. C. Gheller, Dr. A. Guidazzoli, Dr. C. Marchetto), most of all belonging to the same Department.

I also attended the 2nd edition of the CASPUR (Consorzio per le Applicazioni del Supercalcolo per l'Università e la Ricerca) Summer School on Advanced HPC, held in Villa Montecucco, Castel Gandolfo (Roma), from August 28th to September 8th 2006, which was truly inspiring and useful in extending my HPC knowledge and skill. Part of the material present in this report is based on the lectures of this School, so I want to acknowledge the teachers of this School too, Dr. M. Bernaschi (Istituto per le Applicazioni del Calcolo *Mauro Picone*, CNR, Roma), Dr. G. Amati, F. Massaioli, M. Rosati, S. Meloni, L. Ferraro, P. Lanucara and G. Mambrini (CASPUR), Prof. J. Dongarra (Laboratory for Innovative Computing and Dept. of Computer Science, Univ. of Tennessee and Oak Ridge National Laboratory, USA).

2. High Performance Computing (HPC): what does it mean and why do we need it ?

It is nowadays commonly accepted that numerical simulations, based on mathematical models implemented in computational codes run on computing systems, constitute a third type of tools for scientific investigation, along with theory and *real-world* experiments (numerical simulations are sometimes called *in-silico* experiments).

Both *real-world* and *in silico* experiments are characterized by scaling issues: the scales at which you want to realize your experiments require different levels of resources (including money) and technology. The technological improvements of the last decades in the fields of Materials Sciences, Microelectronics, Computer Sciences, have led to the "dream" of higher possibilities in managing numerical simulations scaling issues than the ones in real experiments: as an example, today simulations of magneto-hydrodynamics phenomena of interests for the typical astrophysical scales are implementable, while similar lab experiments not.

This partially-realized "dream" has been, till now, fed by the **Moore's Law**, which synthesizes the leap in computing power in terms of the number of elementary CPU cycles per seconds ("clock frequency"). The improvements in memory technology is at the base of that "dream" too. However, some bottlenecks do exist: the Moore's Law will be no longer valid ten/twenty years from now, because physical limits in the manufacturing of semiconductor devices at the nanometer scale have slowly started to being reached and the promises of new technologies based on optoelectronics, organic materials or quantum information devices seem not realizable on the same industrial scale (at the moment I'm writing this report). Beyond this bottleneck, there is a gap in the performance improvements in CPU and memory technologies. However, this last problem, as we will see below, is partially at the basis of the diffusion of parallel computing, nowadays in the form of Cluster and Grid Computing, pushed forward by the last decade development of new technologies for networking, *in primis* the Internet ones.

Despite these and other bottlenecks, it is commonly accepted, within the scientific and technological communities, the fact that many problems, phenomena and processes can nowadays be realistically studied by HPC and that HPC resources are even more increasing at high rates.

An HPC system, whatever its architecture, can be defined as a computing system that exploits at its near-maximum performance its resources in terms of hardware components (CPU, memory devices, communication buses, etc. ..), their integration and the implementation of algorithms in computing codes. Depending upon the scales and requirements of the problem to be solved, the architecture and resources of the HPC system used can change, but the basic idea of HPC is an efficient exploitation of all the resources of the system: a modern multi-processor supercomputer can't be considered a HPC system without setups targeted to its architecture, components and specific features, otherwise its performance will be drastically reduced.

Top scientific, industrial and business problems requires high level computing resources, other problems, such as playing multimedia movies on desktop PCs, require lower level resources, but in both cases an optimization of the hardware resources is needed.

One possible measure of required resources by a computing job is the total number of basic CPUs operations required for its execution and it is usually functionally dependent on the dimensionality n of the data structures involved in the computation, according to different mathematical relations for different types of problems (power laws, n^{α}, logarithmic laws, $n{\cdot}\log(n)$, etc.). Another measure is the total amount of work (RAM) memory required for the execution of the job.

As an example, I propose here the calculation of the required resources for the execution of a geophysical CFD (Computational Fluid Dynamics) simulation for weather prediction on a time interval ΔT = 24 hours regarding the whole Earth (this calculation was suggested by Dr. G. Erbacci during one of the first lesson at the Cineca Summer School on Parallel Computing, july 2004). This is one of the grand challenges of scientific and technological Research which has not been solved yet, due to technological bottlenecks. Table 1 below considers all the parameters involved in such a simulation.

The model is based on mapping the spherical surface of the Earth on a plane, considering a 2D grid with mesh step of 1 Km. Many (100) such 2D slices are considered, differing for the z-position (radial distance from the center of the Earth, the radius of such mapped spheres are approximately considered constant). Thus, it is a (2+1)D model of the Earth. The 6 variables in the model are: temperatute, pressure, humidity, three cartesian components of the wind velocity vector field. Each time step of iteration in the code corresponds to 30 actual seconds. *flop* means *FLoating point OPeration.* The total number of flops necessary for the calculation, per each time step and grid cell is dependent upon the numerical algorithm for the solution of thermo-fluid dynamics equations on the lattice. Each variable is considered a double precision floating point variable, 8 byte long (according to IEEE 754 standard for floating point arithmetics).

Parameter	Value (dimension)
Earth circumference	$4 \cdot 10^4$ (km)
Earth radius	$6.370 \cdot 10^3$ (Km)
Earth surface	$5 \cdot 10^8$ (Km^2)
# of CFD model variables	6
# of bytes/variable	8 (bytes)
Lateral resolution of 2D cells	1 (Km)
Time step value	30 (sec)
Total # of flop per time step per cell	10^3 (flop)
Total # of vertical slices	100

Table 1: extimates of the basic parameters of a grand-challenge computing problem, the numerical simulation of weather dynamics of the whole Earth in order to predict it on a real-time interval of 24 hours, by the use of geophysical CFD (Computational Fluid Dynamics) code.

Making some simple calculations, it results that the total (minimum) memory required is $2 \cdot 10^{12}$ bytes (2 Tbytes) and a total number of about $3 \cdot 10^3$ steps is needed for a simulation of 24 hours of weather dynamics. The total number of flops required for such a simulation is approximately $1.3824 \cdot 10^{17}$. Considering that the highest peak flops power (according to the special list compiled every six month by HPC communities and presented at the URL www.top500.org) that have been ever reached is 280.6 Tflop/sec (IBM Blue Gene supercomputer, with 131072 processors), the whole simulation will be executed hypothetically in about 8 hours, which is not sufficient (it will leave only 16 hours of prediction per day !).

The total number of flops that can be executed each sec on a HPC system is considered as one of the standard parameters for classifying its performance. Up to 1990s, the main strategy for improving the performance of HPC systems built as massively parallel supercomputers was based on using the top CPU technology developed for the market, on improving the interconnection bandwith between the computing nodes and on rising the volatile and hard memory at their disposal. Today, that strategy is still valid but new approaches have been introduced, based on the idea of using idle time of off-the-shelf common desktop computers distributed in different geographic areas for executing complex jobs divided in small pieces among them. This is the paradigm of Grid Computing, a label which does mean a heterogeneous set of techniques for distributed computing.

While up to 1990s HPC systems were used by specific communities for specific tasks (scientific simulations, large scale enterprise database administration and use, industrial design, etc.) and were usually built as clusters of computing nodes placed in specific buildings, the same technology and methodology has been used for developing similar small clusters or clusters of inter-networked desktop PCs. Nowadays, the HPC techniques and software is diffused among more distinct communities (small-medium enterprises, single Universities or Depts., hospitals, etc.). More attention has been dedicated to the improvement of software performance for exploiting the hardware features of such clusters or PCs.

From the perspective of a scientist with low levels of knowledge in Computer/Computational Sciences and Engineering, such resources, more accessible and distributed, are very useful for small/intermediate simulations. However, it becomes very important to have a little bit of knowledge of optimization techniques in order to adapt one's code to the resources available: a small cluster of PCs can become an intermediate HPC system (with an intermediate cost !) if the simulation codes are configured for best using its hardware components !

Optimization of the code starts at the source code level, goes through the machine code level using tools implemented in modern compilers, then continues with its parallelization for the use on multiprocessor systems. However, optimization procedures must be applied also to serial codes running on single desktop computers and this is starting point for the realization of a HPC code, as I try to show in this report.

3. Computers architectures

At the basis of classical (not quantum) computer architectures, there is a set of logical principles, first of all sequentiality of instructions to be executed by the computer, coded by an algorithm usually represented by a flow chart. Sequentiality of operations has been a strong performance limitation factor since the design of first electro-mechanical and electronic computers. Interaction and communication between processing units and I/O (Input/Output) units, memories and other devices need the execution of multiple instructions in a synchronous way.

The idea of parallel computing is simply based on the possibility of executing different instructions on different or same data at the same time, without excluding the use of a sequential algorithm. Technological and theoretical improvements in the last decades have followed two different ways for realizing parallelism in computing machines:

1. implementation of parallelism in the single CPU, the Central Processing Unit of a Von Neumann's-like sequential computer;
2. implementation of parallelism in a system made of many processing units (many CPUs) communicating data between them.

In the first case, the Moore's law trend (the number of transistors per square inch in microprocessors doubles every 18 months) and the design of new architectures for communication buses between CPUs and memory have pushed forward the performance of microprocessors. In the second case, the improvement in newtorking technology and the development of communication paradigms between computers (and consequent algorithms and softwares) have led to the construction of multiprocessor supercomputers. A combination of both features is at the basis of modern HPC systems.

3.1 The Von Neumann's architecture and its bottlenecks

Figure 1 below schematizes the well know Von Neumann's architecture of a (sequential) computer:

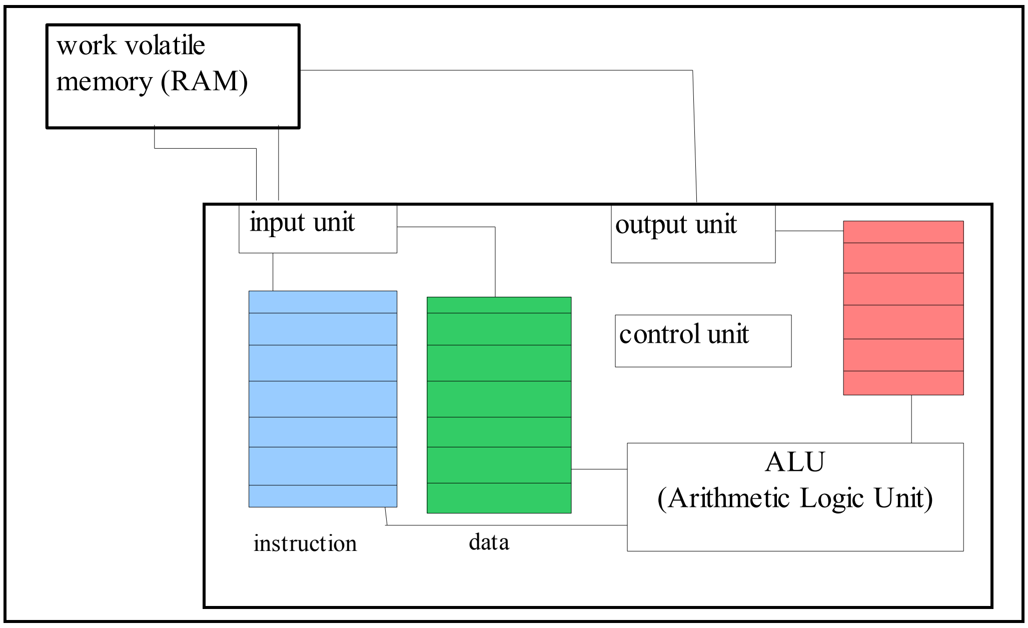


Figure 1: basic logical schema of a CPU according to Von Neumann's architecture; in turkoise blue, the instruction register; in light green, the data register; in salmon red, the output register. Registers are small memory devices implemented within the CPU itself. The ALU is the operational core of the CPU: it can interpret and execute instructions which are arithmetic/logical operations between operands (data in binary format). The input and output units are interface devices between the inner and outer "world" of the CPU, while the control unit has a role of regulation and synchronization of operations.

The basic way of work of such a computer follows the subsequent list:

- fetching an instruction from the work memory (RAM) into CPU internal registers;
- computing the addresses of operands which are data stored in memory;
- fetching of operands from memory to CPU internal (data) registers;
- execution of the instruction (an operation on operands);
- storing of the result in a CPU internal register;
- transmission of the result from CPU storing register to memory.

The first bottleneck of such an architecture for information processing can be summarized as follows: "The instruction stream is inherently sequential – there is one processing site and all instructions, operands, data and results must flow through a bottleneck between processors and memory". Fetching data and instructions within

CPU registers from memory, interpreting instructions, executing them, communicating results to the memory: each of this operation can be executed one at time at each CPU “clock cyle”. The Moore's law trend has led to a powerful increase of the clock frequency: from 100 nsec of the CDC 6600 (1970s) to < 1 ns for Intel Pentium (2000's) for each clock cycle. However, as cited before, the still existing gap between the improvement in microprocessors and memory technologies, the physical limits in raising clock frequency (mainly thermal dissipation, that increases with the increase of the number of devices integrated per square inch) are strong limits to further future technological improvements.

Other issues that influence the performance of sequential computers are:

- the bandwith of the communication buses between CPU and RAM: how many data (bit) per second can be moved from one device to the other; optoelectronics communication devices used within PC promise to raise data transfer velocity, but there are many physical limits, such as the use of semiconductor lasers integrated on-board with other electronic devices;
- the bandwith of the buses for the I/O communications;
- latency times between CPU, RAM and other units, i.e. each device has a different access time, so it is difficult to synchronize their correlated functions and usually it is not possible to use each CPU clock cycle for executing an operation, many clock cycles are used in waiting for data transfer from one unit to another.

Many solutions have been found in the last decade in order to overcome such bottlenecks and this has happened simultaneously with the development of theory and technology of multiprocessor computers. In the next Section, some of this solutions are presented.

3.2 CPU built-in parallelism: technological improvements

Nowadays, the architecture of the CPU of a medium-level desktop PC maintains the basic features of the Von Neumann's one, but it is more complicated, with the addiction of many functional units that have improved speed (clock frequency), reduced memory access latency and realized some forms of parallelism. The top technology available today on the microprocessor market is based on the “**Dual Core**” architecture: two actual identical microprocessors with respective units are integrated in a single package, the clock frequency and control unit are the same for both. Instead of increasing the clock frequency of a single processor, two coordinated processors are used synchronously for the execution of different parts of a job or different jobs. In this way, the thermal dissipation associated to high clock frequency (up to 100 W) is contained and the performance is incremented.

The first Dual Core processor was introduced by IBM with the PowerPC model, in 2003. Intel and AMD, the largest CPU manufacturers of the world, proposed their Dual Core CPUs only in 2005 (Intel Pentium D and AMD Opteron may 2005 and Athlon 64 X2). Both vendors are involved in a large number of projects in order to realize multi-core CPUs, also with heterogeneous cores dedicated to different types of jobs (image and audio signal processing, mathematical calculus).

The Dual Core technology realizes parallel data processing at the scale of the basic unit of a sequential computer, the CPU, but requires dedicated and optimized software in order to exploit its potentialities: with dual or multi-core CPUs multi-tasking and multi-threading (executing more than one program at a time, concurrently, and executing different instances of different programs synchronously, respectively) can be fully implemented, but at the cost of more complicated software (e.g., particular compilers must be used on multi-core processors, otherwise programs are compiled for a single core CPU and only one unit of two is then used). The trade-off between complexity in compiler technology and performance improvement will show us if multi-core CPUs will be successful.

The way towards parallelism in the way of work of a CPU, however, has begun many years ago, with different technological solutions presented briefly below.

3.2.1 Bit-parallel memory and arithmetics

Instead of reading one bit at a time of a CPU word from a memory location (as in the first electronic calculator, which was very sequential), it was introduced by IBM (1950s) a technology for reading blocks of bit from memory locations. Special bit-parallel memories were realized and integrated in mainframe calculators (IBM 704).

3.2.2 Implementation of I/O-dedicated processors

In the first generation of computers, I/O operations were executed by the CPU itself. During the 1950s-1960s, IBM introduced dedicated I/O microprocessors which received I/O instructions from the CPU and then executed them, freeing the CPU from such work. IBM 709 (1958) had 6 dedicated channels for I/O data, while CDC 6600 (1968) had up to 10 dedicated microprocessors that worked independently and asynchronously from the CPU for I/O instructions.

3.2.3 Memory interleaving

Instead of using RAM memories which could be accessed only in a sequential way, a new technology was introduced since 1960s in order to subdivide them in different modules, each of which can be accessed independently from the other. A dedicated electronic circuitry was associated to each module, in order to parallelize the addressing to memory locations.

3.2.4 Hierarchies of memories

In the previous Sections, the gap between memory and microprocessor technology improvements has been cited as one bottleneck of the Von Neumann's architecture of computers: the time for the execution of a single instruction by a CPU is getting lower, the time required for the access of a single memory location by the CPU too but with different rates. Memory technology (both RAM and hard disk ones) has greatly improved in the last decades: modern hard disks can store tens/hundreds of GiBytes data, RAM memories can store up to Gwords; access time to RAM has got from 200 ns of 1980s to 50 ns of DDR models of 1990s. The time for direct access to hard disk is of the order of msec. Transfer of data from hard disk to RAM requires µsec.

Despite technological improvements, the gap still limits the performance of modern computers. A solution for fighting against this limiting factor is buffering the data and instructions transfer from memory to the CPU using a hierarchy of intermediate memory devices which is schematically synthesized in Figure 2. Curiously, this type of solution was proposed for the first time in 1947 by the mathematician John von Neumann himself, but the memory technology did not have at that time the possibility to implement it in a useful way.

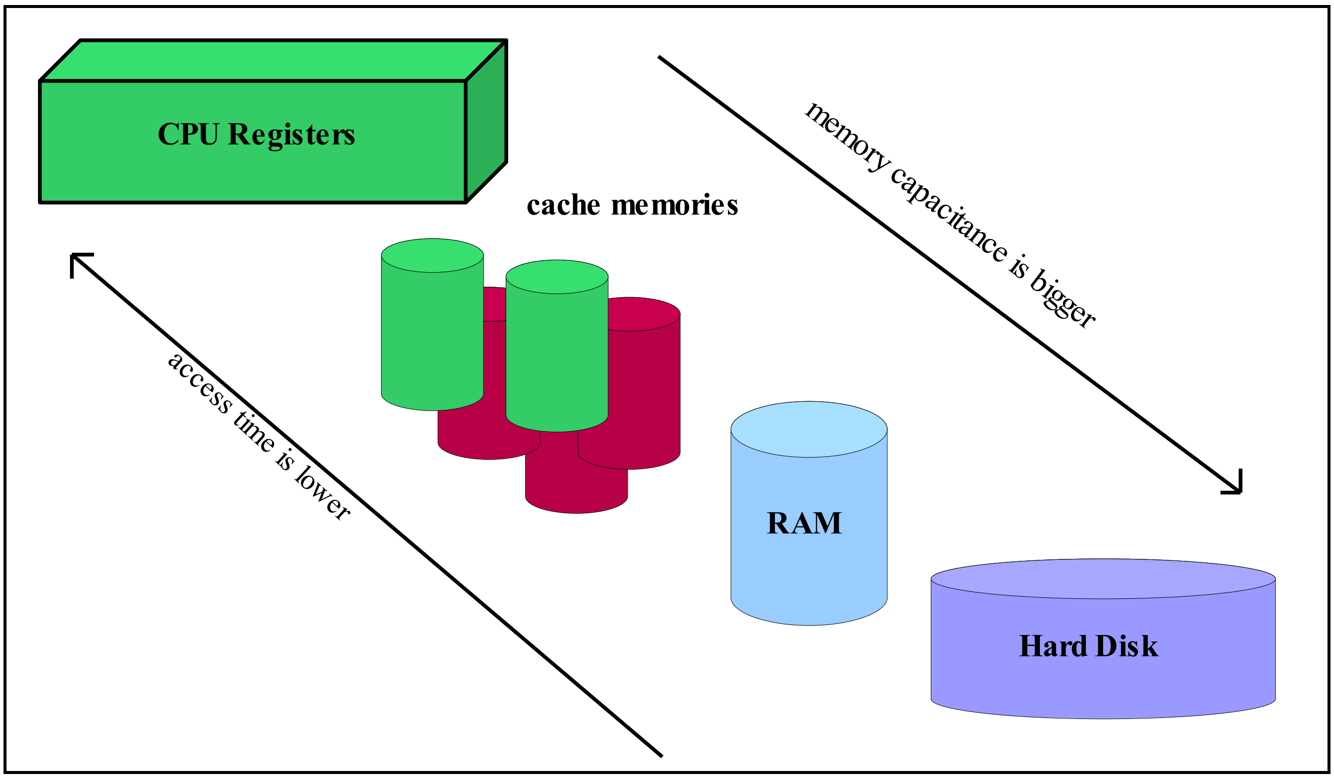


Figure 2: schema of the hierarchy of memory devices between the CPU and the hard disk (non volatile storage device). The cache memory devices are organized according to a specific hierarchy: devices with lower capacitance have lower access time and are "closer" to the CPU, while devices with more capacitance has longer access time.

Buffering data and instructions transfer means that part of them is stored at the intermediate levels between RAM and CPU registers after the first transfer: when they are used again, they are "closer" to the CPU, so time for data communication is reduced. Cache memory devices constitute this intermediate hierarchy and can be on-chip (built within the CPU) or off-chip. They are usually smaller than RAM memories (in terms of capacity) but with lower access time. For example, the DEC Alpha 21164 CPU architecture has a clock frequency of 500 MHz, a level-1 (L1) on chip cache memory with 4 nsec access time, a level-2 (L2) cache with 5 nsec, a level-3 (L3) off-chip cache with 30 nsec and a RAM with 220 nsec access time. The Intel Pentium IV .... *[insert some example of off-the-shelf CPU cache architecture]*

A cache memory device has a completely different architecture in respect of RAM or hard disk devices: it is divided into slots with the same capacity organized as *lines*. *k* subsequent memory locations can be mapped into each cache line. When a data is called from the CPU instruction, if it is not stored in the cache, it is fetched in a cache line location together with its near neighbour data, up to filling in a whole cache line (Figure 3).

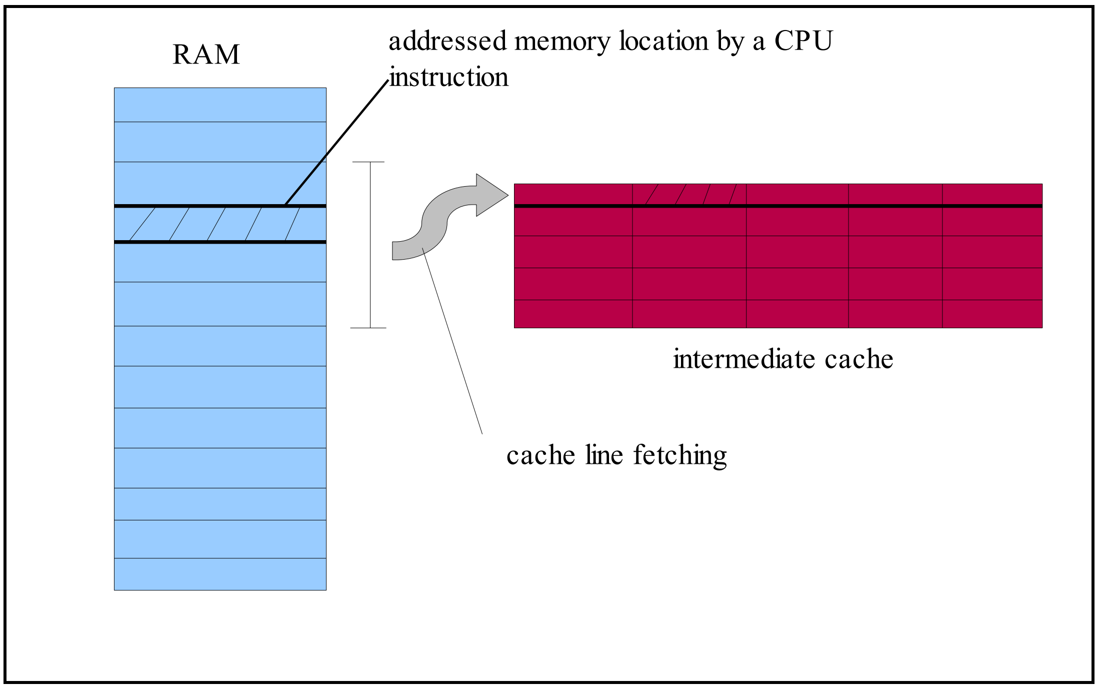


Figure 3: schematic representation of the architecture of a cache memory device and of the cache line fetching mechanism, through which when some data is queried by the CPU, it is transferred from its RAM location into a location of a cache line along with the data that are contigous to it in RAM up to filling in a whole cache line.

When an instruction does modify a data (a variable, for example), if it has been pre-fetched in a cache location, it is updated both in the cache line and in the memory original location, according two possible methods:

- **cache write-back**, *i.e.* each data fetched in a cache line remains therein until the line is requested by an instruction for the memorization of new data; in that case, the updated content of the data is written only in its original memory location;
- **cache write-through**, *i.e.* data is updated synchronously in the cache and memory locations.

The increase in performance of a code depends on cache memories hardware features but also in the way the program has been written: for exploiting the properties of data transfer buffering through the cache hierarchy it is necessary to make use of local references in the source code, i.e. to declare and/or use variables involved in the same expression as closly as possible. There are two types of **local referencing**:

- **temporal**, when an expression or some variables in the source code are used many times at different points of the code;
- **spatial**, when addressing a memory location (for example through a pointer) is followed, in the code, by addressing one of its contiguous locations (for example, expressions involving the use of different elements of an array).

It is shown in the Sections below that the potentialities of data buffering through cache hierarchies are exploited following some rules in designing the algorithms at the base of the source code and using some special optional flags of optimizing compilers, which act at the Assembly and machine level code translation.

There exist both data and instruction cache memory devices: instructions can be fetched into cache lines and the CPU can fetch them from there instead from RAM. This feature is very useful when a code makes an iterative use of the same instructions, diminishing the time of execution because the same instructions are recovered directly from the cache line when needed.

The increase in the performance of a code also depends on the method with which RAM memory locations are mapped into cache lines (see Section 4.2 for the optimization techniques at the source code level in order to take advantage of the cache hardware features). According to this feature, cache devices are subdivided into three groups:

- **direct mapped** cache;
- **fully associative** cache;
- **set associative** cache.

The **cache direct mapping** is the straight-forward way of mapping: subsequent memory locations are mapped into a cache line up to its filling such that every memory block has a direct correspondence to only one cache line. This method is not completely optimal: when the first cells, respectively, of two memory blocks completely mapped into the whole cache are repeatedly required by a CPU instruction, the entire cache must be re-fetched with the appropriate block, the required data may be not immediately at disposal in the cache, then a **cache miss** happens. Cache misses cause a special kind of **overhead** called **cache thrashing**: a whole cache line may be wiped out before it has been accesed only because an instruction asks for an address which is mapped at the beginning of a new set of blocks which completely fill up the cache itself. Among the advantages of direct mapped caches there is the possibility of arbitrary capacity, simple algorithms to substitute values into the locations and a simple method for finding an element in the cache, i.e. to realize a **cache hit** (some bits of the memory address refers to the cache line where that element can be found).

The **fully associative mapping** let a general memory location to be mapped into a general cache line, without following strictly the sequence of data in memory. When a CPU instruction asks for data in a memory location, the corresponding data is firstly searched for in **every** cache line: if it is found, it is transmitted to the CPU registers, otherwise a cache miss signal is transmitted. This mapping method makes use of specific types of cache devices based on an associative memory technology, which is expensive and complicated but necessary for finding an element fetched into the cache. The higher cost of production of a fully associative cache in respect of a direct mapped one limits their memory capacity. The algorithms used for the substitution of data into a location are very complicated too or not efficient: usually a FIFO (First In First Out) approach is used, so the line less used is the one where new data is transferred from the RAM or other cache devices.

Finally, the **set associative** method is a kind of direct mapping replicated on different slots such a type of cache is divided into. This type of cache device is usually made of 2 or 4 slots (two-way or four-way set associative). That means that a memory block can be mapped in a cache line on a slot or in the corresponding cache line of another slot (2-way associative). This method and corresponding technology let data located in different memory locations to be mapped into a similar cache line but in different slots, in order to avoid cash thrashing overhead. This type has been introduced in order to balance between the complexity of electronic circuits needed for the fully associative approach and the low fault-tolerance of the direct mapping one.

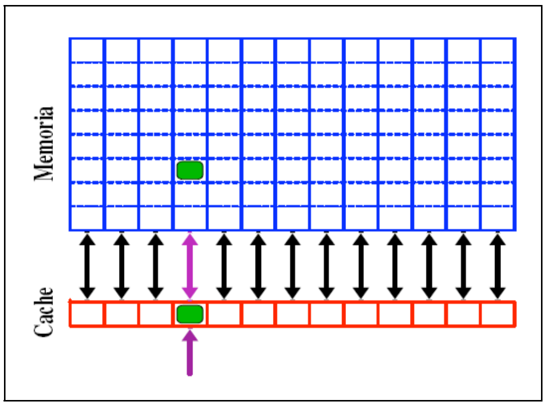
Usually, all cache memory devices are set associative: L1 cache 4÷8 way set associative; L2,L3 2÷4 way set associative or sometimes direct mapped in order to have more capacity. Figure 4a, 4b, 4c show schematic representations of the three respective different methods of mapping between RAM memory and a cache device.

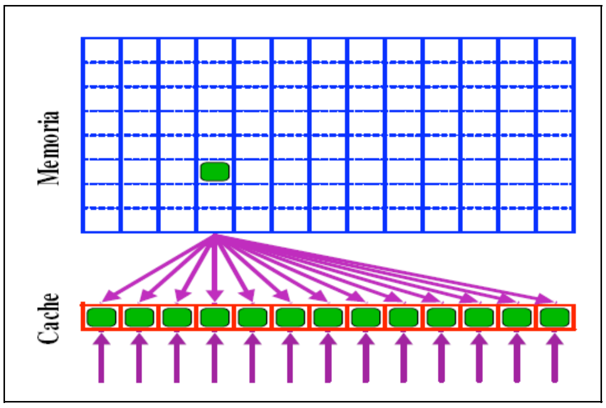
3.2.5 Multitasking

Multitasking has been one of the most important technological solution introduced in CPU functionality in order to go beyond the strict sequential model of the Von Neumann's architecture: the execution of a job is made by a sequential set of instructions, i.e. computing and I/O operations; during the I/O operations the CPU is idle, waiting for data communication; instead of leaving it idle, the CPU can be charged with instructions of another job. This is a first example of built-in parallelism: many pieces of different jobs (programs) are executed simultaneously. A particular type of multitasking implementation is **time sharing**, that is the concurrent execution of different programs by different users letting each user to interact with his/her-own process (CPU bursts of activity are used for each program).

3.2.6 Instructions look-ahead

Instruction look-ahead is another technological solution developed for breaking the strict sequentiality of Von Neumann's architecture: the decodification and memory localization of an instruction is made ready for the CPU before it needs or requires it according to the flow of the algorithm. In this way, the performance of the CPU with multiple function units is improved (see subsection below). This methodology is implemented at the Assembly code level by the activation of optional flags of compilers, which reorganize the structure of the source code into a new one.





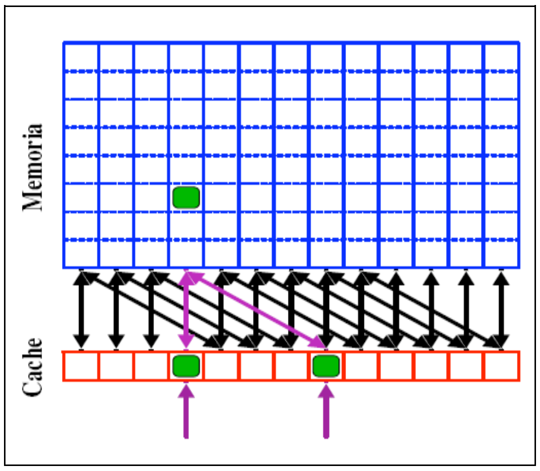


Figure 4 a) top-left, direct mapped cache memory: each RAM memory block is mapped only into a specific cache line. Different RAM blocks can be associated to the same cache line and this may lead to frequent cache misses or cache trashing. b) top-right, fully-associative cache memory: there is not a unique correspondence between a RAM block and a cache line, every block can be mapped nto every cache line. This model may be the optimal one, but is very technologically complicated and expensive, with restricted capacity. c) bottom-left, 2-way associative cache memory: an intermediate type between the previous two, a memory block can be mapped into two different cache lines which are correlated because there is a fixed "separation" (slot, in terms of cach lines) between them. Courtesy from G.Amati, F. Massaioli, Optimization and Tuning – 2, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

3.2.7 Multiple function units

A leap toward CPU-level built-in parallelism has been the subdivision of CPU components in different devices dedicated to specific functions that can be executed indipendently. For example, in older computers, the CPU contained general registers (data and instructions), special registers (e.g. the program counter), the control unit and the ALU (Arithmetic Logic Unit). This last microprocessor was responsible of any type of calculation, one at a time. In modern CPU architectures, the ALU is divided into distinct microprocessors dedicated to distinct functions (boolean algebra calculations, fixed and floating points computing, etc.). The compilers are able nowadays to translate a source code in an optimized object code whose instructions flow is structured such that different calculations are made by the different units. However, the exploitation of this type of parallelism implies a correct use of the compiler, which must be customized to the specific CPU architectures. Modern compilers, as the reader can find below, are targeted to different architectures using optional flags.

3.2.8 Pipelining

The pipelining is a general method of work subdivision and management, implemented in a CPU: a task T^i (flow of instructions) is divided into many sub-tasks T_j^i (different sets of instructions or single instructions); the flow is divided into different stages S_k too; at each clock cycle, only a sub-task can be executed. At time step 1 (1st clock cycle), T_1^1 is executed within S_1; at time step 2 (2nd clock cycle), T_2^1 is executed within S_2, but instead leaving S_1 idle T_1^2 is executed within it, and so on at subsequent clock cycles. In this way, the task T^1 can "flow" through the different stages S_k of the CPU without leaving idle the stages, which begin executing sub-parts of other task T^i. It should be noted that there is a sequentiality in the sub-tasks execution: T_j^i can not start if T_{j-1}^i has not finished.

The schema of a CPU or dedicated microprocessor which implements pipelining is represented in Figure 5.

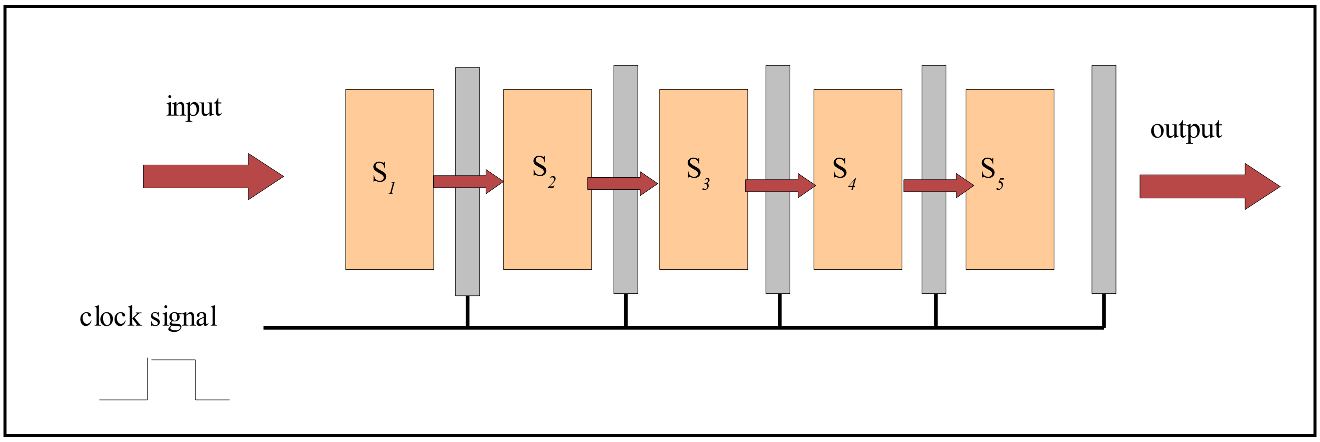


Figure 5: logical schema of a CPU implementing pipeling. The pipeline is made of a sequency of elaboration devices (stages S_k). Each stage makes calculations (boolean or arithmetic) on different parts of data/instructions. Between each couple of stages, there is a **latch** device which acts as a kind of register for local intermediate data to be passed from a stage to the other (the output of a stage is the input of the subsequent). Data/instructions flow into the pipe according to a clock signal which regulates the execution of a sub-task in a stage and the transmission of data from a stage to another.

Pipelining is a fundamental methodology for reducing idle time of microprocessors and is very useful in linear algebra arithmetics (array arithmetics), because it requires the execution of the same operation (for example a sum of floating point variables in the case of the sum of 2 1D floating point arrays) many times on different sets of operands. If each stage runs a part of that operation, at each clock cycle (except the first and the last) such parts are executed on respective components of the different arrays.

3.2.9 Floating point arithmetic: SIMD technology

Most common off-the-shelf PCs CPUs belong to the x86 (or 32 bit) architecture class, also called IA_32 refering to Intel 32 bit architecture (one of the most famous and diffuse). 32 bit architecture means that the basic “words”, with which instructions are codified, are 32 bits long.

Since 1970s, computers have been more and more charged with new types of data elaboration involving multimedia data (signals, images). Computer Graphics has started to be improved since then. Multimedia processing and Computer Graphics usually require the execution of the same type of operation (mainly a mathematical one) on a huge amount of data of the same type. For example, in Computer Graphics, operations such as geometric transformations need to be applied to large sets of homogeneous data (fixed or floating point variables representing vertices). In multimedia applications, a common operation is the change of the brightness of an image (i.e. a 2D map of values). Three variables are associated to each pixel of an image, indicating the level of red, green and blue colors, respectively, according to a common scale. To change the brightness of the image, a same value must be added or subtracted to each variable of each pixel (grid point in the 2D map).

A technique and corresponding technology to realize the execution of the same operation on a large amount of data of the same type using only one instruction was developed along the 1970s and 1980s for dedicated parallel supercomputers (vector and array processors) (see Section 5 for the Flynn's classification scheme) and for dedicated microprocessors or functional units, for example DSPs (Digital Signal Processors) or GPUs (Graphics Processing Units): it is called the **SIMD** (**Single Instruction Multiple Data**) technology. The same principles and corresponding technology have been implemented in off-the-shelf CPUs in the last decade, with different solutions, in order to satisfy first the needs of multimedia and graphics processing.

In 1997, Intel introduced in the desktop PC market the first architecture with SIMD capabilities, the Pentium **MMX** (**MultiMedia eXtension**). These extensions use new commands to perform two, four or even eight integer instructions in parallel, which are commonly over-used in image processing. The MMX architecture uses the **FPU** (**Floating Point Unit**, one of the functional units cited in Section 3.2.7) to load the data adjacently into registers, using 64 of the 80 bits of each FPU register (8 per CPU, also known as x87 registers). In this way, the programmer has to decide between common x87 or MMX instruction sets for floating point arithmetics and this can be done by inserting Assembly language instructions in the source code.

In 1999, Intel released on the market the Pentium III architecture, including 8 new dedicated registers and 70 new Assembly language instructions to perform single precision floating point computation using SIMD techniques. Such extension is called **SSE** (**Streaming SIMD Extension**) technology. Each new register is 128 bits long and is labeled xmm*i*, where *i* = 0,1, ..., 7. In 2000, Intel introduced the first versions of Pentium IV with a new extension of SIMD features, **SSE2**, with a set of new instructions for handling with double precision floating point calculations, exploiting the power of the xmm*i* 128 bits registers, each of which can contain 2 double precision FP numbers. In this way, the applications requiring massive FP computing, such as those of Computer Graphics or Scientific Computing, has found place also on off-the-shelf desktop PCs.

AMD, the other more important CPU and microprocessors manufacturer, has developed its own SIMD extensions to the x86 architecture and has called it **3DNow!**. This technology adds (single precision) floating point capabilities to MMX architecture, extending the Assembly instruction set.

MMX, SSE, SSE2 (and successive new extensions, nowadays called SSE3 and SSE4 in the last Pentium IV versions) and 3DNow! have their own Assembly instruction set. Each specific instruction can be **scalar** or **packed**: scalar instructions are the common Assembly ones applied to the least significant element of a vector, which can be contained in the large 128 bits xmm*i* registers; packed ones simultaneously apply to all the elements of the vector. Figure 6 illustrates a scalar and packed multiplication of single precision FP variables, which can all be stored contigously in a SSE register.

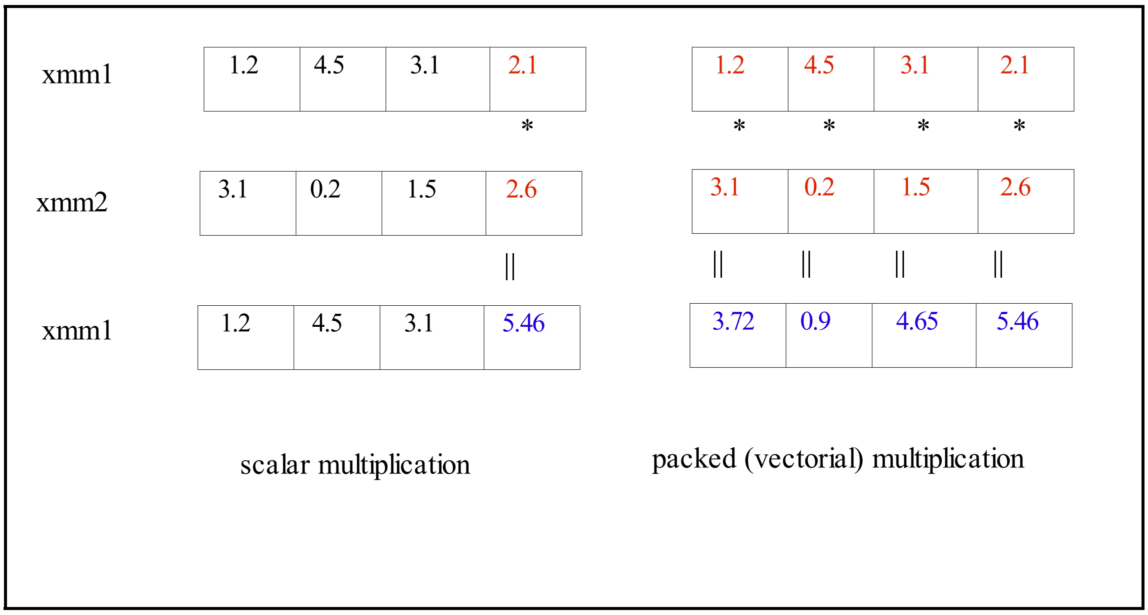


Figure 6: schematical representation of scalar and packed SSE multiplications. 2 SSE registers are used for the two sets of operands (in red), xmm*1* and xmm*2*. In this case, the SSE multiplication is made between single precision FP numbers, coded as 32 bit long strings. Each SSE register can contain 4 such values, that is a 1D array of length 4. The scalar multiplication is a multiplication between the last elements of the respective arrays (registers) and it produces a new number stored in the same register position (in blue), while the packed multiplication is a per-element operation, so it produces 4 new values.

Despite the possibility of vectorial (so parallel) floating point arithmetics operations, there are some bottlenecks in the use of SSE/SSE2 instruction sets: the programmer must pay attention to how the program organizes data into memory, because loading data into SSE registers requires they occupy contigous memory blocks that start and end on 16 bytes borders (**data alignment**). This means that software development must be tuned to SIMD architecture, otherwise the usual FPU instruction set and registers are used for floating point arithmetics: a Pentium IV then makes floating or fixed point calculations with the same performance of a Pentium or 80386 !

High level programming languages do not specify standardized SIMD functions. SSE-aware object code relies on the capabilities provided by the compiler or on the possibility of introducing Assembly language instructions within the source code. In the first case, the code can be **automatically optimized** for SIMD instructions by invoking special option flags of the compiler. Most of modern compilers have these capabilities: the dedicated flags are reported in Section 4.3 for GCC and in Section 4.4 for the Intel compiler. In the second case, there are some languages, such as C/C++, that permit to introduce Assembly instructions in the source code, so a **manual optimization** can be realized. Figure 7 illustrates a piece of C source code in which a function for the scalar product of two vectors is defined. The source code presents both a usual implementation (left column) and a modified version (right column) containing Assembler instructions which do refer to SSE registers. In the second case, the programmer must be a little bit aware of CPU Assembly language and instruction set in order to use them.

```
01 //Compute scalar product
between two vectors in C
02 float
scalarproduct(floatx[],floaty[])
03 {
04 return x[0]*y[0] + x[1]*y[1]
+ x[2]*y[2] + x[3]*y[3];
05 }
```

```
01 //Compute scalar product
between two vectors in C
using Assembler instructions
02 float
scalarproductAssembler(float
x[],float y[])
03 {
04 vector temp;
05 asm(
06 “movl   %1,      %%esi;”
07 “movl   %2,      %%edi;”
08 “movaps (%%esi), %%xmm0;”
09 “mulps  (%%edi), %%xmm0;”
10 “movaps %%xmm0,  %0;”
11 :”=g”(tmp)
12 :”r”(x),”r”(y)
13 );
14 return tmp.f[0] + tmp.f[1]
+ tmp.f[2] + tmp.f[3];
15 }
```

Figure 7: comparison between two C source code typescripts defininf a function that returns the scalr product of two vectors. The left column typescrit is a typical C code: the output of the function requires 4 additions and 4 multiplications between single precision variables. The right column typescript shows the definition of a similar function which makes use of Assembler instructions within the `asm();` function call: line 6 means “move the first operand in the register “`esi`””; line 7 means “move the second operand in register “`edi`””; line 8 means “move the content of register “`edi`” in the register “`xmm0`””; line 9 invokes the `mul` instruction executed in packed mode (`p`) and on single precision floating point data (`s`), so `mulps`; the instruction is a multiplication element by elemeny of the data in the two registers. Than teh result of the packed multiplication is placed in register `xmm0` and then moved to register `%0`. The structure `tmp` of type `vector` then receives the output of the `asm();` function. As can be seen, 4 additions are still needed, but only one instrcution for the multiplication, a packed instrcution instead of a scalar one. Courtesy from [Siemen2003b]

Manual tuning of the code to the CPU architecture, using Assembler instructions, is very difficult, leads to a complicated source code and requires technical knowledge of the Assembly language of the used CPU. Scientists using computers for computing may not have such knowledge or possibility to obtain it. However, manual optimization is the best way in order to adapt your code, piece by piece, function by function, for using completely and in a performant way the hardware resources. For these reasons, CPU manufacturers, such as Intel, usually provide **intrinsics**, that is special data types and functions, collected in libraries, which are C compiled versions of Assembler instructions. This is an intermediate solution between the use of pure Assembler and the use of automatic optimization by compiler. This last solution is more user-friendly but it usually makes the code less perfomant. However, it is the most widely used in the Scientific Computing communities and for this reason it is presented in Sections 4.3 and 4.4 for two different compilers. It should be noted that modern compiler manufacturers (Intel, Portland Group) or developers (GCC initiative community) have been increasing the number of optimizing flags at disposal of the user for many types of architectures. The important thing is a correct exploitation of such flags by the user when he/she compiles his/her programs and this implies that he/she must know a little bit of the hardware architecture of his/her desktop PC (or parallel supercomputer) and must make an extensive use of the manual of the compiler used. These requirements are not so considered by scientists who use computing facilities as tools for obtaining scientific results.

3.2.10 CPU design “philosophies”: past and future sceneries

The previous list of theoretical and technological solutions for improving the performance of CPUs must be considered in a more general framework of basic design philosophies developed and implemented in the last decades by the Microelectronics and Computer Science Research communities and manufacturers.

3.2.10.1 CISC and RISC architectures

CPU architectures are usually classified as belonging to 2 different families: **CISC (Complex Instruction Set Computer)** and **RISC** (**Reduced Instruction Set Computer**).

The **RISC** architecture is the one used in the most diffused microprocessors and microcontrollers (i.e. for embedded computers) and is based on a philosophy that favours a simpler set of instructions (at the Assembler level) which all take about the same time to execute.

In the early days of the computer industry, CPU designers tried to make instructions do as much work as possible: for example, one instruction that would do all of the work in a single operation, i.e. loading up two numbers to be added, adding them, then storing the result back directly to memory. At that time, there were two types of reasons for such a design philosophy, called **CISC**:

- compiler technology did not exist, programming was done in either machine code or Assembly language; to make programming easier, complex and integrated instructions that implemented a whole set of successive operations were very useful;
- memory devices were, at that time, very expensive, small (in term of capacity) and slow (in terms of access time), it was advantageous to store information in computer programs instead of memory.

The complex instructions of first CISC CPUs were direct representations of corresponding high level functions of high level programming languages. They were inspired by a general principle called "orthogonality principle": to provide every possible addressing mode for every instruction. Beyond this, it should be added that during 1960s-1970s, built-in CPU registers were very expensive and difficult to be integrated on chips, so CPUs had to communicate mainly with RAM devices: decreasing the frequency of access to RAM was a must, so complex instructions seemed to be a good solutions to such issue.

However, in the late 1970s, researchers (mainly at IBM) demonstrated that many features of such CPU instruction sets, designed for faciliting coding, were ignored by real world programs and the instructions themselves took several processor cycles to be performed. In those years, advances in theoretical Computer Science were leading to new technologies in writing compilers. Such early stage compilers did not take advantage of such instruction sets. Most of these instructions were not used frequently by developers, so CPU designers did not care tuning them and the result was that they were slower than a number of smaller (but simpler and tunable) operations doing the same things.

CISC architectures were complicate to be implemented into hardware and did not leave empty space for registers and built-in caches, memory devices which were being introduced in those years and which were even more useful for facing the bottleneck of frequency clock-RAM access time gap. Many analyses showed during the 1970s and 1980s that very used constants in common programs would fit in 13 bits, while almost every CPU design of that time dedicated some multiple of 8 bits for storing them. So, in order to decrease memory access frequency, instructions might be implemented with a smaller number of bits, leaving room for constants coding. Other studies confirmed that real-world programs spent most of their time executing very simple operations, so the most used ones should be implemented as simple and fast instructions: the goal, at that time, was 1 instruction in 1 clock cycle.

These issues led to the RISC design philosophy, which has one advantage over the CISC one (less need to use registers or memory space because data can be embedded in instructions) and one disadvantage (a simple task is run as a series of simple instructions, but the total number of instructions read from memory is larger, therefore takes longer).

Some technological advances towards built-in parallelism (pipelining, embedding multiple function units in a single chip, see Sections 3.2.7 and 3.2.8) introduced in the 1980s seemed to favour the RISC approach, because the core logic of a RISC CPU requires fewer devices and leaves more free room for other devices.

In fact, many academic, then industrial, projects were devoted to the development of RISC architectures. The most famous ones were:

- DARPA VLSI project;
- UC Berkeley's RISC project (1980), which led to the RISC-I and RISC-II prototypes, this last one being used as the basic model for nowadays RISC microprocessors;
- Stanford University MIPS program (1981) which led to the MIPS design used for commercial applications in Computer Graphics consoles (PlayStation and Nintendo 64) and common embedded processors for high-

end applications;
- IBM Power (1990s-2000s) and PowePC (2000s) processors, which are the most famous RISC processors used on mainframes, IBM supercomputers and dedicated embedded microprocessors (e.g. for the automotive industry).

Despite its succesful characteristics and tendency to support the integration of the CPU built-in parallelism solutions, the RISC design philosophy has not managed to become the standard in each field of microprocessors design: the CISC architectures still are the most diffused in the desktop PC, high performance supercomputers (except for IBM's ones which make use of Power technology) and commodity servers markets. While RISC processors dominate the market of mainframes and of embedded CPUs and controllers, the x86 architecture (a CISC one) remains the *de facto* standard for desktop PC, for three main reasons:
- the very large base of proprietary PC applications are written for x86, whereas no RISC platform has a similar installed base and this has meant that PC users have been locked into x86 despite lack of performance;
- RISC architectures have been able to scale up in performance quite quickly and cheaply, but Intel and AMD (the most powerful microprocessors manufacturers) have taken advantage of their large markets for spending enormous amounts of money on processors development and improvements;
- Intel and AMD designers has managed to apply some features of RISC design philosophy to x86 architectures: for example, the P6 core of Intel PentiumPro and its successors have special functional units that expand and "crack" the majority of CISC instructions into multiple simpler RISC-like operations.

The resources of Intel and AMD in supporting their x86 products have been so high and uncomparable with the ones of pure-RISC manufacturers that today x86 CPUs are more performant than RISC ones, except for IBM Power and PowerPC processors, which are good competitors.

3.2.10.2 VLIW (Very Long Instruction Word) CPUs

The **VLIW** architecture was first proposed in th early 1980s by J. Fisher at Yale University. It describes an instruction set philosophy in which the compiler packs a number of simple, non-inter-dependent operations into the same instruction word. When fetched from cache or memory into the processor, these words are easily broken up and the operations dispatched to independent execution units. This approach to CPU architectures can be described as a software- or compiler-based superscalar technology.

Nowadays, the most diffused CPUs with a VLIW architecture are the ones belonging to the **x86_64 family** or generally the 64-bits processors. Up to 2005, the 64-bit CPUs have been used on servers, multiprocessor supercomputers like clusters of PCs and also desktop PCs. The first 64-bit CPU was introduced in 1991 by MIPS. In 1994, Intel announced the successor of its x86 technology, the **IA-64** architecture, effectively released on the market in 1998-1999 with the Intel Itanium. In the same year, AMD announced its 64-bit solution, which is now known as the x86_64 architecture, copied by Intel in 2004 with its **EM64T** platforms. In this case, AMD was the winner in the run towards 64-bit CPUs for desktop computers, with the AMD Opteron and AMD Ahtlon 64. Also IBM has dveloped some lines of its PowerPC processors using the 64-bit technology.

64-bit architecture first of all means that the integer registers of the CPU are 64 bits long. These registers can contain the values of pointer variables, used for addressing the RAM or the virtual memory. In a 32 bit architecture, the amount of memory space that can be addressed is no more than 4 GiBytes. This has been an unreachable amount of memory up to some years ago. Multimedia, scientific computing or large-scale database applications today require many GiBytes of real or virtual memory and such memory devices are now available on the common market place.

A CPU with 64 bits long words (coding of instructions) is also more performant in pointers, fixed point and floating point arithmetics. However, it requires dedicated drivers, compilers and operative systems (OSs). One major problem with its introduction in the market place is the compatibility of software, mainly written for 32 bit architecture. Most x86_64 processors guarantees a hardware-implemented compatibility with x86 software, because the 32 bit instruction set is still supplied with the new one.

3.2.10.3 SMP (Symmetric Multiprocessor) and Multi-Cores computers

Section 3.2 is dedicated to technological solutions for implementing different forms of parallelism within the CPU itself. Although the described technologies and methodologies have led to performance improvements which could not be imagined years or decades ago, many physical limits have been reaching in the integration of semiconductor devices in single chips. It seems that CPU built-in parallelism has been pushed forward as much as possible, so computers manufacturers are trying to extend multiprocessor architectures typical of supercomputers also to desktop and server ones.

A **SMP (Symmetric MultiProcessor)** computer is an hardware architecture very well known in the field of parallel computers: on a single board, many CPUs are implemented, sharing the RAM memory. Such an architecture is also called **shared-memory parallel computer** (see Section 3.3.2.1) and it has been shown that its performance scales down with more than 16 processors, due to the difficulties in the management of memory access by the different units. A SMP architecture requires dedicated software, written with specific tools for implementing multi-threading programming techniques.

Many expectations have arisen in the last two years about new technologies which let the integration of two CPUs, with their respective inner caches, in a single chip/package. In such a way, two processors can function at the same clock frequency, increasing computational power without increasing power dissipation (which are nowadays of the order of 100 W) and device temperature. Such an architecture has been called **dual-core** and was first introduced in 2003 with the IBM PowerPC models. Both Intel and AMD released their first versions of dual-core processors in 2005 and their plans for the future are addressed towards **quad-core** processors (4 CPUs in a single package). There are also projects for heterogeneous multi-core processors, where each core is a CPU dedicated to specific type of data processing (image processing, audio processing, mathematical computing, etc.). Desktop computers based on dual-core processors are yet on the market: Intel Pentium D and AMD Athlon 64 X2 and FX60 are the last ones released by the two manufacturers in 2005-2006. Such computers require OSs specifically developed for their architecture and dedicated compilers and this seems to be the major drawback at the moment for their diffusion, also in the HPC field. However, they are symptomatic examples of the future plans of CPU manufacturers: new forms of parallelism are investigated and searched for, in order to overcome the limits inherited by the Von Neumann's architecture.

3.3 Parallel (multiprocessor) computers

A **parallel computer** has been theoretically defined as *a **large collection of processing elements that can communicate and cooperate to solve large problems fast***. According to this definition, a single modern CPU can then be considered as a parallel computer due to its architecture based on multiple function units (see Section 3.2.7). However, the processing elements cited above are the CPUs.

Parallel computing is a computing approach devoted to solve a single problem (through the execution of a program) subdividing it in many homogeneous or heterogeneous sub-problems to be solved by the single CPUs (processing units) simultaneously. The complete original problem solution is guaranteed by the **communication beetween the different processing units**.

A classification scheme of computer architectures was proposed in 1966 by M. J. Flynn [Flynn1972] and it is still used also for introducing the different types of parallel computers architectures. This scheme is based on two criteria: the distinction between **instruction stream** (sequence of instructions to be executed) and **data stream** (sequence of data over which instructions act); the molteplicity of hardware used for processing instruction and data streams. According to these criteria, there are four possible types of computer architectures resulting from the combinatorial association of different ways of instruction and data stream management. Figure 8 shows the management types and the result of their combinations.

The SISD architecture corresponds to the classical Von Neumann's one: a scalar computer with only one CPU without possibility of pipelining. The MISD architecture has never been extensively implemented, apart from some small Research initiatives. The SIMD architecture have been already considered in Section 3.2.9 regarding the implementation of some of its principles to single CPU architecture, but there we mention that these ones were inspired by multiprocessor architectures. Finally, MIMD configuration is the one more diffused today in the field of supercomputers.

3.3.1 SIMD systems

Figure 9 shows the basic schema of a SIMD multiprocessor computer, implementing a special kind of parallel computing model called **synchronous parallelism**.

Two main types of SIMD supercomputers were manufactured in the last decades:

- **array processors**, made of a set of identical processing units operating simultaneously on different datasets; they were commercialized during 1970s-1980s and were very performant with vectorial calculations (each processing unit executes the operation on a component of the vector); some of them were realized as independent machines, while other types were modules to be integrated in other computing environments (**attached array processors**);
- **vectorial supercomputers**, characterized by an extended instruction set containing vectorial

instructions to be applied to array-like data sets; they had specialized functional units able to implement vectorial operations, through the exploitation of pipelining (see Section 3.2.8); one of the most successful was the Cray 1, developed in 1976 by S. Cray.

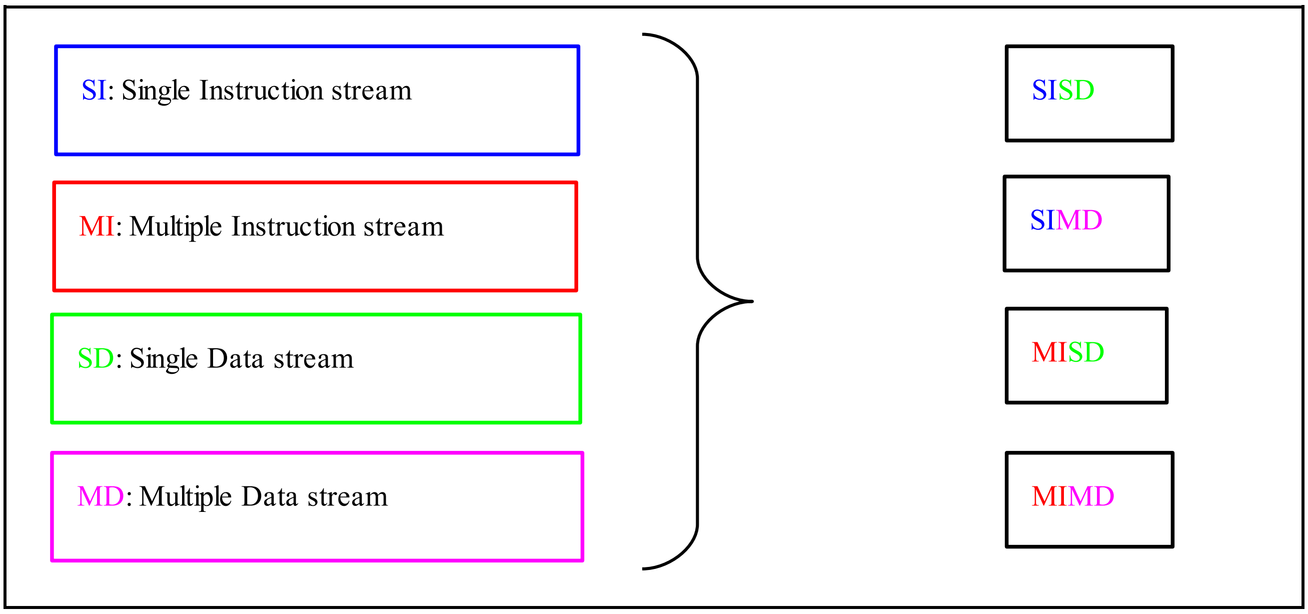


Figure 8: representation of the four different types of computer architectures according to the Flynn's schema [Flynn1972]. The four categories are the result of all the possible combination of two different types of data stream management and two other distinct types of instruction stream management. The MISD model has never been implmented at the industrial scale, only some academic projects focused on certain issues tried to realize such a type of CPU.

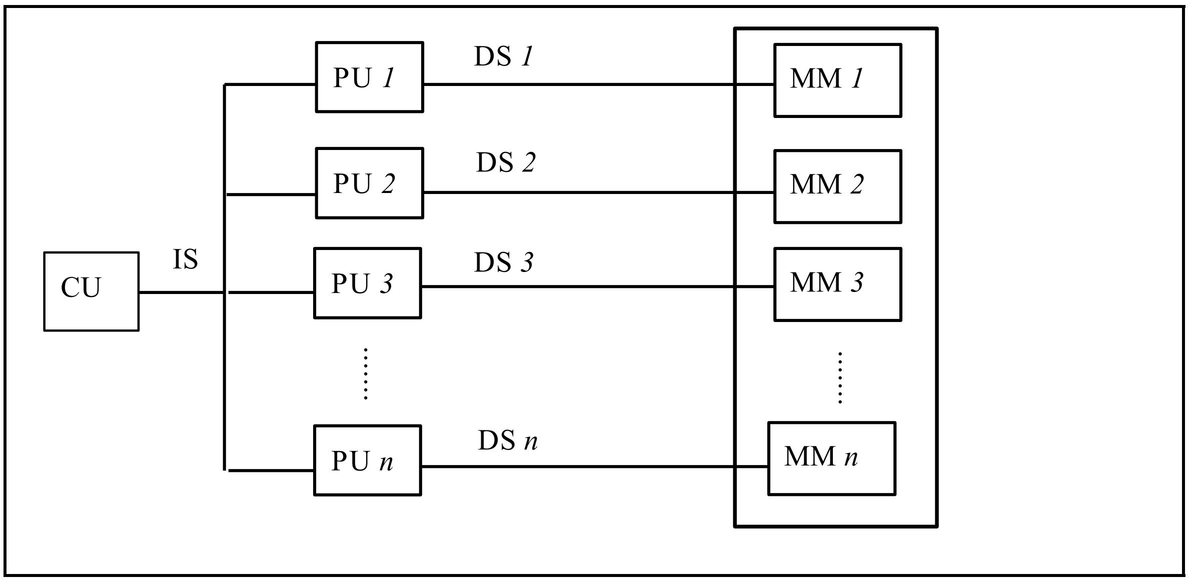


Figure 9: basic general schema of a SIMD multiprocessor supercomputer. A single Control Unit (CU) distributes a single Instruction Stream (IS) to *n* Processing Units (PUs) (in general CPUs). Each PU executes that sequence of instructions on a different Data Stream (DS), i.e. on a different data set obtained from a distinct Memory Module (MM). In this way, a single operation is simultaneously applied to different data (**synchronous parallel processing**).

Among the most performant SIMD parallel supercomputers, we should aknowledge:

- the **Connection Machine** by Thinking Machines, which, in its 2nd version, has 64K processors; it was released in 1987; each processor was directly connected to a local memory; groups of 32 processors might optionally share floating point accelerator units; this supercomputer was famous for its esterior design, for the possibility of arranging blocks of processors according to specific types of interconnection topologies corresponding directly to topologies of data structures on which parallel programs needed to operate;

during the 1980s and beginning of 1990s it was a leading example of supercomputer, appearing also in scenes of science-fiction movies like Jurassic Park;
- the **APE** massively supercomputers, developed by an initiative of the Italian National Institute for Nuclear Physics (INFN) and commercialized during 1990s by QSW (Alenia S.p.A) under the name Quadrics; the project APE100 led to a parallel supercomputer with a peak power of 100 Gflops, which was a significant level for those years.

3.3.2 MIMD systems

MIMD (Multiple Instruction Multiple Data) computers were the further development of the paralllel SIMD ones. During the 1970s, the first big multiprocessor mainframes entered the market (Univac 1100/80, 1976, Burroughs B-7800, 1978, Cyber 170/720, 1979, IBM 3081, 1980). They still were affected by residual bottlenecks deriving from the Von Neumann's architecture: only the management systems of the multiprocessor architecture obtained a performance leap, because the number of jobs completed per unit of time increased. However, the usefulness of multiprocessing consists in the possibility of using independently or correlately the CPUs at the same time (parallel processing) and not assigning to each CPU a different job to be executed in a certain intrinsic time different from job to job.

MIMD architectures are theoretically divided into
- **multicomputer systems**, i.e. a set of interconnected but autonomous computers, communicating between them for exploiting parallel processing;
- **multiprocessor systems**, i.e. a system where many CPUs share together the same work memory.

Nowadays, supercomputers are hybrid architectures, i.e. at certain scales units can be individuated and classified as multiprocessor systems, but at higher scales these units are integrated within a multicomputer architecture. The first class is also labelled **distributed memory architecture**, the second one **shared-memory architecture**.

Both types (or hybrid ones) supercomputers can exploit parallel processing, according to different paradigms which will be discussed in Section 5.1. Since the beginning of 1990s, new types of supercomputers have been classified and proposed within the class of distributed-memory systems: they have been called "**computing grids**" in analogy to energy transport grids. Multicomputer systems are usually tailored for scientific, industrial or business-management applications, located in a single building, their computational units are CPUs with their RAM memories integrated in large boards called chassis. Each chassis can contain many CPUs and centralized hard disks shared between them. Many chassis are then collected in a single "armchair" (see Figures 10-11 for a typical example of distributed-memory supercomputer), a **cluster supercomputer**, thus called because many single computers are "clustered" in a single case. Computing grids, however, are made of single computers (sometimes off-the-shelf desktop PCs) interconnected by LAN or WAN or Intranet and located in different rooms of a building or in different buildings or in different regions of the Earth. There is no complete agreement in HPC communities (both industrial and academic) on the terminology to be used: some computing grids are called clusters of PCs. However, two main differences characterizes the two types of distributed-memory systems:
- traditional cluster supercomputers have homogeneous computing nodes (i.e. chassis, sets of identical CPUs), interconnected by dedicated internal networks;
- computing grids usually are made of heterogeneous computing nodes, for example different PCs, thus the interconnection system is of more general type, a LAN or directly the Internet.

3.3.2.1 Shared-memory architectures

Multiprocessor architectures are generally made of a limited number (from 2 to 32) of very powerful CPUs that share the same RAM memory. The memory access is coordinated and uniform: each CPU is characterized by the same access time for a generic memory location. The interconnection between memory and CPUs is realized through three different types of systems: common bus, crossbar switch or multistage network. Each CPU may have a local cache.

The SMP (Symmetric Multiprocessor) architecture described in Section 3.2.10.3 is an example of shared-memory supercomputer. Such machines are sometimes called **tightly coupled systems**, because all the resources of the whole system are shared between the different CPUs. This type of architecture may be subdivided into two different sub-sets:
- **UMA** (**Uniform Memory Access**) architecture, when each CPU sees a common space of memory addresses and has the same access time to it (as previously said); this is the typical architecture of a SMP; if the implementation of this architecture implies that when a CPU updates the content of a memory location the whole system updates the values contained in each location of a cache memory device of a CPU

mapped into that memory location, then that machine belongs to the sub-category of **Cache Coherent UMA** (**CC-UMA**) SMPs;

- **NUMA** (**Non Uniform Memory Access**) architecture, when many SMPs are interconnected between them such that each CPU of a SMP can have direct access to the RAM memory of another SMP, with different cross access times, which are obviously larger than the CPU's access time to its local RAM memory; also in this sub-case, some implementations are classified as Cache Coherent NUMA machines, due to the same mechanism of global updating of cache memories of each CPU of each SMP.

For the parallel programming of such small supercomputers, see Section 5.1.2.

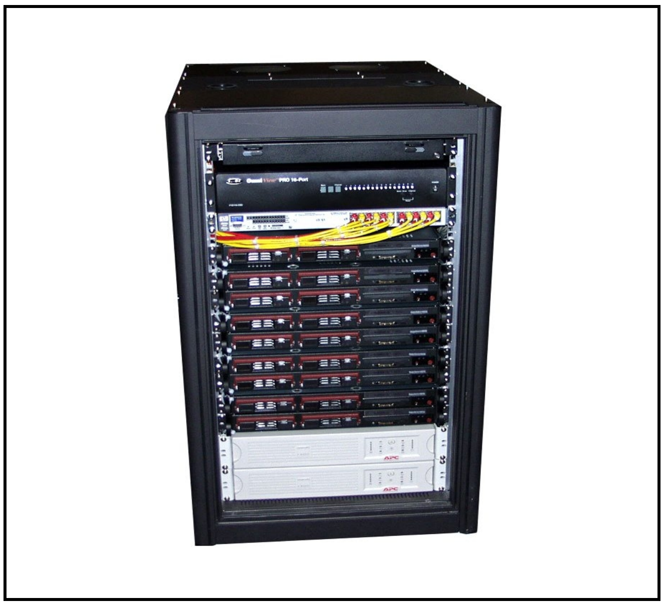


Figure 10: an example of small cluster supercomputer, made by 8 chassis each containin a “computational node” made of 2 AMD Opteron 242 CPUs sharing a memory module of 1 GiBytes. Each computational node is a small SMP and is interconnected with the other by a dedicated 1 Gigabit internal network. This hybrid cluster computer is installed at the Bioinformatics and High Performance Computing Lab of the Bioindustry Park of Canavese (*http://www.bioindustrypark.it/*).

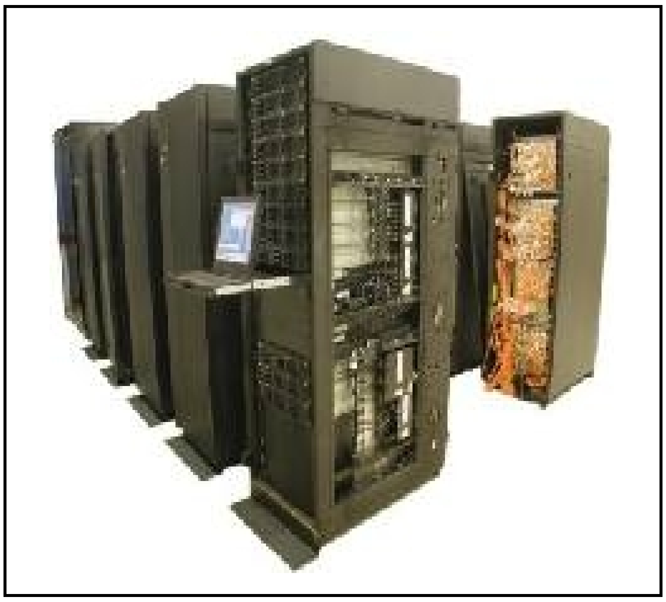


Figure 11: an example of a large scale cluster supercomputer installed at Cineca. It is an IBM Linux Cluster 1350 made of 512 IBM X335 computational nodes. Each computing node contains 2 Xeon Pentium IV CPUs that share together 2 GiBytes of work memory. Each black “armchair”, also called rack, contains many chassis, each chassis contains a certain number of computational nodes.

3.3.2.2 Distributed-memory architectures

The multicomputer architecture, also called distributed-memory system, is based on the clusterization of different processing units (CPUs) each of which has its local RAM memory. Therefore, such architecture is also labelled the **NORMA** (**NO-Remote Memory Access**) model: each CPU directly accesses its own memory locations and the other CPUs cannot have access to it. The functioning of the supercomputer as a whole is guaranteed by a protocol of communication of data and instructions between the processing units through a model called **message passing**: messages are input into the network that connect the units and routed towards the correct destination.

Usually, distributed-memory supercomputers have a large number of CPUs (from tens to hundreds of thousands) but each of them is not the best performant on the market. For example, homogeneous or heterogeneous Linux clusters of the type Beowulf are made with common off-the-shelf CPUs or sometimes with old ones off the market. In this case, the high degree of parallelism is the asset of the architecture.

3.3.2.3 Hybrid architectures

As anticipated in Section 3.3.2, nowadays parallel supercomputers are hybrid systems in the sense that they are built by assembling subsystems of different types. Thus, the concept of **computational node** acquires importance in describing the logical (high level) architecture of the whole machine. A computational node can be a smaller supercomputer, built according to a specific setup: it can be a single CPUs with its RAM, cache and hard memory devices, as in heterogeneous clusters of PC or it can be a SMP system sharing RAM and hard memories (in this case the whole supercomputer is defined as a **cluster** (or **federation**) **of SMPs**) or it can be a smaller distributed-memory system.

Taking as examples the parallel supercomputers at Cineca (up to July 2006), different architectures by different manufacturers can be shown:

- the IBM SP5 is an IBM SP Cluster 1600 supercomputer, made of 64 computational nodes of the type p5-575 interconnected with a pair of connections to the Federation HPS (High Performance Switch); each computing node is a SMP, so a shared-memory supercomputer, with 8 CPUs of the type IBM Power 5 at 1.9 GHz; 60 nodes have 16 GiBytes of memory each, while 4 nodes have 64 GiBytes each; globally the supercomputer has 512 CPUs; the bandwith of the HPS network is 2 Gbit/s; this is an example of hybrid system, made as a federation of interconnected SMPs; the 4 nodes with more RAM memory are suitable for shared-memory programming, while the general machine can be considered as a distributed-memory system considering as elementar units the computational nodes, so message passing programming is used for parallel simulations; more information about such architecture by IBM can be found at the URL http://www.ibm.com/servers/eserver/pseries/library/sp_books ;
- CLX is an IBM Linux Cluster 1350 made of 512 IBM X335 computational nodes interconnected between them through a Myrinet network with maximum bandwith of 256 Mbit/s between each pair of nodes; each node is a 2-way SMP containing 2 Intel Xeon Pentium IV CPUs; each node has 2 GiBytes RAM memory; the total number of processors is 1024, 768 of which are Xeon Pentium IV at 3.06 GHz with 512 MiBytes L2 cache while the other 256 are Xeon Pentium IV Nocona models (64-bit technology of Intel, called EM64T, alternative to the x86_64 standard) with 1 GiBytes L2 cache and support for the Hyper-Threading Technology; this is an example of the flexibility of cluster supercomputers, because they let the system manager and owner to assembly an heterogenous federation of computational nodes, such as in a single supercomputer different levels of performances can be reached by users; indeed, this is very useful for large machines such this one installed at Cineca, accessed by many users which have different necessities and priorities; such flexibility and scalability let also the owner and system manager update the hardware of the whole supercomputer without the need to change its basic architecture and setup;
- XD1 is the third parallel supercomputer installed at Cineca offered to scientific communities; it is a Cray XD1 model which collects 72 computational nodes in only one cabinet; the nodes are divided into 12 chassis and each of them is a SMP made of 2 AMD x86_64 Opteron CPUs at 2.4 GHz; each node has 4 GiBytes of memory.

Beyond this type of hybrid architectures (a parallel supercomputer as a distributed-memory system of computational nodes, where each node may be a SMP, i.e. a shared-memory system made of CPUs), there are other types commonly used today with special models of memory management:

- **NUMA** (**N**on **U**niform **M**emory **A**ccess) supercomputers, which belong to the category of distributed-memory systems because RAMs are physically distributed among the different nodes, but the ensemble of local memories is setup in order to realize a global memory space addressable by each CPUs; such architectures have dedicated hardware devices for realizing such common address space; the access time to a memory location is dependent on the physical distance between CPU and location (the fastest access is obtained on the respective local memory) and on the bandwith of the inteconnection system (used with accesses to non local memories); NUMA shared-memory supercomputers, cited in Section 3.3.2.1, are sometimes considered also as hybrid machines, i.e. distributed shared-memory systems;
- **COMA** (**C**ache-**O**nly **M**emory **A**ccess) supercomputers, which are similar to NUMA systems except for the fact that the distributed memories are used for a large common cache space accessible from each CPU through directories.

While in pure distributed-memory systems the only type of communication between CPUs is realized by message passing (see Section 5.1.1), in hybrid systems or pure shared-memory systems there are different (not exclusive) ways through which CPUs have access to memory resources:

- through a **physical share**, i.e. the CPUs have the same addressing space;
- through a **logical share**, i.e. a software middleware lets two different processes running on two different CPUs have access to the same data structures simultaneously.

A SMP, for example, may implement both a physical and a logical share of memory, while a pure distributed-memory systems none of them, therefore message passing is the only way for sharing data between CPUs. The message passing model of communication can be used also within systems that adopt physical but not logical memory share: message passing is simulated through shared buffers. In distributed-memory systems such as NUMA, which have a logical share, a special kind of programming model called **LINDA** is used, which lets the user to implement a sort of virtual distributed memory space made of tuples (data structures).

3.3.3 Interconnection issues

Despite the case of SIMD supercomputers, which can only execute the same job on different datasets instead of subdividing a single job into many sub-jobs run on different computational nodes, an important component of a MIMD supercomputer is the interconnection network which is the medium through which multiple processors are connected each other and to memory devices.

An interconnection network can be **complete** (each computational node can communicate directly to each other one) or **indirect** (there is a pattern of connections between nodes which usually has a particular topology). The first case has never been implemented for two main reasons:

- the cost of the network is proportional to n^2, where n is the number of nodes;
- most parallel algorithms usually involve small parts of a network.

Many different types of interconnection topologies have been used in different parallel supercomputers, with different types of topologies described by graphs. The topology is very important in the design of a MIMD parallel computer because its performance depends upon many features which derive from the chosen pattern of interconnnections: for example, **latency time** (the delay on the network that occurs between addressing the node of arrival and forwarding the data packet to it), **bandwith** (the amount of data that can be sent through the network per unit of time), **scalability** (the possibility of adding nodes to the architecture increasing the performance of the machine), **reliability**, all of them are influenced by the chosen topology in respect of the architecture of the system.

The simplest topology is a linear one, realized with a single bus all nodes are attached to. It is a very unperformant topology, originally used in SMP systems. It is used also in cluster of desktop PCs connected through a LAN. Although its cost is very low and optoelectronics nowadays offers large bandwith on LAN cables, it has a limited data transmission rate, it is not scalable and the performance rapidly decreases when too many nodes are sending or receiving data through it.

The role of interconnection topology in HPC is still present at each scale of a supercomputer architecture, i.e. at the scale of interconnections between computing nodes and at the scale of connections between CPUs within a single computational node (like in a SMP).

3.4 Grid Computing

In Sections 1 and 3.3.2 the labels "Grid Computing" and "computing grids" are used for refering to the new frontiers of HPC technologies and methodologies. In Section 3.3.2, it is cited the fact that the term "Grid Computing" is used within different semantic contexts and there are no common standard definitions of what Grid Computing resources are.

Although this report can not treat the issues connected with Grid Computing, it is interesting to have a rapid view of this field of ICTs and its role in future trends of HPC, because it is strictly connected to the history of HPC.

Grid Computing is considered nowadays one of the most important frontiers of research in the field of parallel architectures. As a field of Research & Development in University and industry, it is concerned on a basic idea, very succesfully comunicated to the world by I. Foster and C. Kasselman in their famous book of 1999 [Foster1999]: in the future, computing resources might be shared in such a way to let any organization of people to ask for an amount of such resources optimal for the task to be accomplished. It is the paradigm called "On-Demand Computing", a core business of IBM, formulated upon the similarity to energy demand: when a user needs energy for a device, he/she connects the device to a transport grid through which energy is shared.

The technologies needed for the realization of such a paradigm are information processing grids, generally called computing grids. The leaders of the Globus project (one of the most ever funded till now on such subjects) define a computing grid as "a persistent environment which let realize software applications with the aim of integration of computing, data management, visualization, remote control of instrumentation resources belonging to different administrative domains, sometimes geographically distributed".

Such a definition of computing grid implies that they can be distinguished into four different types of grid [Migliardi2004]:

- **pure computational grids**, as aggregation of computing resources belonging to different security and management domains (for example different institutions or social subjects located in different parts of the world);
- **data grids**, as aggregation of storage and management resources belonging to different domains;
- **application services grids**, the natural extension to the Internet of the concept of ASP (Application Service Provider), i.e. resources that can be rent on a remote server for running applications for certain times;

- **instrumentation grids**, as aggregation of tools for making (scientific) experiments or run processes that can be controlled remotely.

As reported in Section 3.3.2, pure computational grids are then natural evolution of parallel architectures of the type cluster supercomputers, which still are shared computing resources but within a single domain. Clusters of PCs interconnected through a LAN are examples of machines closer to the idea of computational grids, but with a main difference: they are statically defined, the maximum number of resources is fixed and also the interconnection pattern. A user can exploit the resources of all the computers or of only a subset of them, but those computers are the fixed resources available to the cluster. During the 1990s, in industry and academy, the first projects for building distributed-memory supercomputers using heterogeneous computing nodes made of off-the-shelf hardware led to software libraries such as PVM (Parallel Virtual Machine) and networking solutions in order to overcome one of the most critical bottleneck of Grid Computing, the necessity of giving to the user an integrated image of the grid as a whole system, despite its heterogeneity. New ideas and advancement in software let the realization of middlewares devoted to this integration, realizing what is called a **SIS** (**Single System Image**).

Such advancements led from cluster architectures as the Beowulf one (the most know in the field of cluster supercomputer, from a project begun in 1994 at Stanford University) to the first examples of computing grids made of nodes geographically distributed, such as the ones used by the SETI@home or the folding@home projects. These projects are based on the idea of voluntary sharing resources of common desktop PCs, through the Internet technologies, for running identical tasks (programs) on different datasets, in order to find patterns within large-scale datasets. The key component of such computing grids is the software, not the hardware: a simple screen-saver installed on the desktop PC which can benefit of the idle CPU time for running its task and communicating results to a server.

While the Beowulf architecture is the traditional example of a cluster supercomputer which can not be considered a computing grid according to the previous definition (due not only to the homogeneity of its units but mainly because it belongs to only one specific domain), the previously cited initiatives are common examples of first types of actual computational grids, whose resources dynamically change in time according to the number of users who decide to adhere to the project itself. They are good examples also of how computing grids can be programmed, according to the technologies today available: they use a “farmer-worker” model, in the sense that a server (the “farmer”) creates a set of tasks (a “bag of work”), when a computing node (a PC) declares its disponibility on the network (through the Internet) it receives a task, executes it and returns back the output to the server. This model of “parallel processing” is useful when the tasks are independent from each other, as it happens with similar programs to be executed with different input.

There exists another type of computing grids programming model, based on the definition of such a grid as a set of HPC systems (for example a federation of cluster supercomputers). In this case, each task is a parallel program realized through the standard methods used on cluster supercomputer (see Section 5) with the addition of instructions to redirect the task to one node instead of another according to the availability of resources requested by the task.

At the moment there are no other models for executing jobs on computational grids. The third type of computing grids cited above (application service grids) are at the moment futuristic and matter of academic research: their main objective is the aggregation of independetly developed applications for the realization of new super-applications. Such a computing grid implies that it is made of a data grid and a pure computational one.

Finally, data grids and instrumentation grids are natural evolution of Internet technologies and they have proven to be more realistically implementable than computational ones, as demonstrated by the huge number of projects dedicated to their realization.

4. 1st level of HPC: optimization of scalar codes (squeezing the hardware of your desktop PC !)

Despite the cases where the order degree of memory requirements, the number of nodes or elements (according to the fact of using a finite differences or elements method), the number of operations needed for the total execution imply the need of a parallel supercomputers, for smaller and faster jobs nowadays commodity computers can be perfomant enough. They can be even more performant if the programs are tuned to their hardware features, in order to exploit all the advantages deriving from technologies described in the previous Section.

The tuning should be addressed expecially to the CPU and memory features and ways of operation. It may be useful to remind the difference existing between the **architecture** of a CPU and its **physical implementation**: the architecture consists in the Assembly instruction set and in the integer, floating point and state architectural registers; the physical implementation includes the physical CPU registers (usually 2.5÷20 times the number of architectural registers), the CPU clock frequency and the time of execution of each instruction, the number of functional units (for integer arithmetics, logic bitwise operations, floating point arithmetics, address computation, memory *load&store*, branching prediction and execution, etc. ...), the dimensions, the numbers and the hierarchy of cache memories, the units for the realization of the **Out Of Order Execution** (**OOOE**) and/or the **Simultaneous Multithreading (SM)**. One architecture can be realized with distinct physical implementations: for example Intel's Pentium III, Pentium IV, Xeon, Pentium M, Pentium D, AMD's Athlon and Opteron are all implementations of the same architecture, the x86 (32 or 64 bits) one; IBM's Power3, Power4, Power5 are different implementations of IBM's Power architecture.

The architecture of a CPU is more "trasparent" to the programmer than its implementation. However, the performance of a code differs strongly from one implementation to another. This fact leads directly to two basic initial rules for improving the performance of a code:

1. it must be compiled on the computer it is going to be run;
2. the programmer must tune the code not only according to the architecture but mainly according to the CPU's features and memory implementation.

For scientists who do not want or could not spend too much time in training themselves in HPC methods and approaches, there are some general rules, techniques and compiler features to be known without too much fatigue for obtaining a significative improvement of the performance of their code. This type of tuning is even more fundamental if the scientist would like to realize a parallel version of his/her own code in order to scale up the dimensions of the problem or computation he/she needs to do: an optimized scalar code is the best starting point for writing its parallel version, otherwise the parallel code will not scale in performance increasing the number of processors used.

In the Section below, the reader can find a small number of suggestions, tricks, rules for optimizing his/her code. As said in Section 1, the compiler of reference is GNU **GCC** (**GNU Compiler Collection**) **gcc** (for C language) or **g++** (for C++ language). Analog treatments can be done for GCC Fortran or other programming languages compilers.

Optimization of a code can be done at different scales, with different objectives (obtaining a high speedup of the program without care of its memory requirements or balance the tradeoff between memory needed and speed of execution or realizing a program with good performance for different types of implementations of an architecture instead of a program with higher performance but less portability) and by different tools. Section 4.2 introduces some useful and quite general techniques which improves the performance of a code in terms of use of memory and speed of execution. Such techniques can be directly implemented by the programmer when he/she writes the source code, although many modern compilers can realize them too. The programmer must need to know a little bit about of the used architecture in order to exploit them. Section 4.3 is dedicated to optimization made automatically by the compiler. Modern compilers are very powerful and might be considered as "expert systems" in translating a source code into its machine-readable version. They can implement different types of optimization:

- optimization independent from the programming language;
- optimization dependent upon the language;
- optimization dependent upon CPU architecture;
- optimization dependent upon CPU and memory implementations;
- optimization for a better use of cache memory hierarchy;
- optimization based on prediction of what the program will do during runtime.

The first three categories are overlapped with the types of optimizing techniques which can be implemented by the programmer itself. This fact means that the scientist who could not or do not want to learn some details about the architecture used is free to leave the compiler to act instead of him/her-self. Usually, it might be dangerous because the compiler is an "intelligent software" but can not match all the requirements for a useful optimization, so it is usually advised to learn such basic methods presented in Section 4.2 in writing a scientific computing code.

The other categories of optimization can be realized only through the compiler, unless the scientist is also an expert in software engineering and hardware or has a special interest in hacking his/her own code.

The first step towards optimization consists in understanding how the code works during runtime. So we consider here that the code has already been debugged using the tools of GCC compilers.

4.1 Profiling your programs

The term "**profiling**" usually means obtaining information about the features of the program execution (how the program's parts behave during its runtime). Such information can be obtained at different scales in respect of the source code structure:

- profiling at the **subroutine level**, i.e. information about the execution flow of the code, the functions calls and their behviours;

- profiling at the **single statement level**, i.e. information about a single code line or expression.

Profiling is based on sampling data about the runtime behaviour of the program, usually through three different techniques:

- **TBS** (**T**ime **B**ased **S**ampling), i.e. the state of the program counter (a special CPU register tracking the address of the instruction being in execution at that CPU cycle) is sampled at fixed times in order to understand which point of the code is being executed;
- **CBS** (**C**all **B**ased **S**ampling), i.e. counting the number of times a single subroutine/function is invoked and the time spent within it each time, without considering the data passed to the subroutine/function itself;
- **EBS** (**E**vent **B**ased **S**ampling), i.e. counting the number of events happening during runtime by the use of special hardware counters.

On a per-function basis (CBS), the GCC offers a program called **`gprof`** that records the number of calls to each function and the amount of time spent therein. Functions which consume a large fraction of the run-time can be easily identified from the output of such program. So, the efforts to speed up a program should concentrate first on those functions which dominate the total runtime.

It is necessary to activate a flag of GCC compiler when compiling and linking the source code. In the command line of compilation (or within a Makefile) the **`-pg`** option must be added. This option leads, after compiling and linking, to an **instrumented executable program** which contains additional instructions that record the time spent in each function. It is important to stress the fact that if a program consists of more than one source file then the **`-pg`** option should be used when compiling each source file and used again when linking the object files together to create the final executable. Forgetting to link with the **`-pg`** option is a common error, which prevents profiling from recording any useful information.

While running such an instrumented executable program, profiling data is silently written to the binary file **`gmon.out`** in the same directory where the program is placed. These sampling and writing activities are responsible for an overhead which depends on the structure and features of the code itself. That means that profling is an intrusive measurement of the performance of the code and this is an important feature to be remembered.

After the run, the **`gmon.out`** file can be analyzed, invoking the **`gprof`** command with argument equal to the name of the executable file. Figure 12 below shows an example of report generated on the `stdout` by the command **`gprof <executable program name>`**. The first column of the data shows that the program spends most of its time in the function called `step` and the remaining time in the `nseq` one: efforts to decrease the run-time of the program should concentrate on the former. In comparison, the time spent within the `main` function itself is completely negligible (less than 1%). The other columns in the output provide information on the total number of function calls made and the time spent in each function.

```
$ gprof a.out
Flat profile:
Each sample counts as 0.01 seconds.
 %     cumul.    self             self   total
time  seconds  seconds    calls us/call us/call name
68.59    2.14     2.14 62135400    0.03    0.03 step
31.09    3.11     0.97   499999    1.94    6.22 nseq
 0.32    3.12     0.01                          main
```

Figure 12: example of a report on the performance of a code made by the GNU program `gprof`, which is invoked at the prompt of the shell with argument equal to the name of the program. The example is taken from [Gough2005].

Figure 12 is an example of *flat profile*: each subroutine/function is analysed independently from the others. **`gprof`** can generate also *call tree profile*, where each subroutine/function is considered along with the ones it calls and the report distinguishes between the time spent in the subroutine/function itself and the time spent in called subroutines/functions within it.

Further or more complicated information about functions calls can be obtained using the **`gprof`** command along with specific options. For example **`-l`** option generates the line by line profiling, **`-l -A -x`** options are used for the same type of report plus the listing of the code. TBS profiling can be useful for the individuation of most frequently accessed lines and which of them are slower. As previously stated, such amount of information is obtained at the cost of an instrumented code, different from the original one and, perhaps, with different performance.

For a general view of the potentialities of this program, but also of its limitations, consult the manual *GNU* `gprof`*. GNU Profiler* by J. Fenlason and R. Stallman, freely available at http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html or at the official WWW site of the GNU `binutils` package, http://sourceware.org/binutils/docs-2.17/gprof/index.html, which **`gprof`** belongs to.

The programmer should be very careful in reading and interpreting **`gprof`** reports when

- the code makes use of many external libraries;
- a subroutine is invoked many times (different runs can require different execution time, but **`gprof`** measures only the cumulative time of their execution);
- the code is made by only one subroutine/function (no useful information in this case);
- the code has been optimized by the compiler through the inlining technique (see Section 4.2).

Another GCC tool that can be used for profiling a code at the single line level is the program **`gcov`**, which can analyse the number of times each line of source code is executed during run, thus making it possible to find areas of the code which are not used or not exercised in testing. This type of post-run analysis is called **code coverage** and it needs each source code file being compiled along with the options **`-fprofile-arcs`** and **`-ftest-coverage`**: the first one incorporates instrumentation code for each branch of the program, while the second one adds instructions for counting the number of times individual lines are executed. Branch instrumentation records how frequently different paths are taken through `if` statements and other conditionals. The coverage data are then created when the program runs, stored into several files with the extensions `.bb`, `.bbg` and `.da` in the same directory of the program. As for **`gprof`**, this data can be analyzed invoking the command **`gcov`** followed by the name of the executable program. The **`gcov`** command produces an annotated version of each source code file, adding to the file name the extension `.gcov`, containing counts of the number of times each line was executed. In such new source code files, the lines which have not been executed are marked with hashes ######. More information about the program **`gcov`** can be found at the URL http://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Both **`gprof`** and **`gcov`** work only with programs compiled with GCC compilers.

Understanding how the workload is distributed among different functions is one of the first steps for addressing the optimization actions to specific parts of the code. Software developers also use coverage testing in concert with testsuites, to make sure software is actually good enough for a release. Testsuites can verify that a program works as expected; a coverage program tests to see how much of the program is exercised by the testsuite. Developers can then determine what kinds of test cases need to be added to the testsuites to create both better testing and a better final product.

It must be remembered that the code test coverage requires the source files not to be compiled with optimization options, otherwise the actual structure of the code will be different from that written by the programmer (see Sections below about optimization).

4.1.1 Example of software for performance analysis

Within the Open Source communities, many software tools have been developed for offering integrated environments for debugging and profiling programs. I do not want to enter into the debugging field, which is very complicated but also full of support tools. Taking into account only the profiling step of testing a program, I would like to suggest another software package which should be very useful for both debugging and profiling. Its name is Valgrind and is a bundle of tools for debugging and profiling programs written in any language, although C/C++ programs may get more advantages than other ones. Valgrind runs on x86/Linux, AMD64/Linux and PPC32/Linux, several of the most popular platforms in use. It debugs and profiles your entire program: unlike tools that require a recompilation step, it gives you total debugging and profiling coverage of every instruction executed by your program, even within system libraries. You can even use it on programs for which you don't have the source code. Valgrind is extensible: it consists of the Valgrind core, which provides a synthetic software CPU, and Valgrind tools, which plug into the core and instrument and analyse the running program. Anyone can write powerful new tools that add arbitrary instrumentation to programs. This is much easier than writing such tools from scratch. This makes Valgrind ideal for experimenting with new kinds of debuggers, profilers, and similar tools.

Valgrind is suitable for any type of software. It has been used for almost every kind of software project imaginable: desktop applications, libraries, databases, games, web browsers, network servers, distributed control systems, virtual reality frameworks, transaction servers, compilers, interpreters, virtual machines, telecom applications, embedded software, medical imaging, scientific programming, signal processing, video/audio programs, NASA Mars lander vision and rover navigation systems, business intelligence software, financial/banking software, operating system daemons, etc, etc. See a list of projects using Valgrind.

4.2 Source-code level optimization

"Optimization is a complex process: for each high-level command in the source code there are usually many possible combinations of machine instructions that can be used to achieve the appropriate (same) final result. The compiler must consider these possibilities and choose among them".

The previous sentences, cited from the GCC gcc/g++ manual by B.J. Gough (chapter 6), synthesize the idea of machine-driven optimization: advanced optimizing compilers may translate the source code into the machine code in such a way to adapt its low-level structure to the architecture and way of operation of the computer. Modern compilers satisfy such requirements. However, the optimization is not left only to the machine: the programmer should make good choices in writing his/her codes for two reasons:

- it is true that the compiler can translate the same source code into different machine codes with different structures more or less performant on the specific computer architecture implementation, but the compiler is not completely an "intelligent machine", it applies pre-defined rules according to the arguments (options) passed to it, if a source code is written according to complicated and unuseful algorithms rather than in more efficient or adapt ways, its performance will not gain too much from automatic optimization; that's the reason for knowing a little bit of the hardware the programmer is going to use;
- the design of the algorithm and its efficiency is left to the programmer, although many new software tools have been developed for assisting the programming activities; the same result (the execution of a set of operations) can be obtained with different patterns of instructions or (logic) flow structures, adapted to the way the CPU works.

Two basic common examples of optimizing rules that can be applied at the source code level are:

- **Common Subexpression Elimination (CSE),** i.e. trying to process an expression with fewer instructions, by reusing already-computed results;
- **Function Inlining (FI),** i.e. substituting the instruction that calls a subroutine/function with the "body" of the subroutine/function itself.

An example of use of the first technique, in C language, is showed in Figure 12 below: instead of making the CPU repeat the same instructions for calculating two times the same expression with the same variables (memory locations), that expression is stored in a "swap" (intermediate) variable `t` used for computing the final expression.

CSE is very powerful in speeding-up the code but has many drawbacks so its exploitation should be considered case-by-case. Some examples of CSE side effects:

- there are subroutines/functions that at each call update the values of global variables or built-in constants, e.g. the C function `rand()` update the index within the pseudo-random numbers list whenever it has been called in order to return a new pseudo-random number at the next call; it means that it should not be substituted in any expression otherwise the pseudo-random number will be always the same; the same observation stands in the case of multiple calls of subroutines/functions with the same arguments values but with inter-dependent calls;
- the use of swap variables increases the use of registers (**register spilling**) with the consequence of possible slowing down of the execution.

Many compilers are able to implement CSE automatically, without the necessity of swap variables: sometimes it is sufficient to delimitate recurrent expressions with parentheses, they recognize the expression and do not calculate it again if it has been done previously in the code. This automatic implementation of CSE can be dangerous in the cases cited above: with the use of parentheses or with option at compiling time, automatic CSE can be silenced (`-nostrict` option with `gcc`).

FI is very useful when a subroutine is invoked many times in the same part of a code, for example within the "body" of a loop structure: whenever a function is called, a certain amount of extra time is required for the CPU to carry out the call itself, because it must store the function arguments in the appropriate registers and memory locations, jump to the begin of the function (bringing the appropriate virtual memory pages into physical memory or CPU cache memories if necessary), begin the execution of the code and then return to the original point of execution when the function call is complete. All this additional work (except for the execution of the body of the subroutine) is refered to **function-call overhead**. Function Inling eliminates this overhead by replacing calls to a function by the code of the function itself (the code is placed in-line).

The subdivision of a program into subroutines/functions calls is useful for the design of the code, because it realizes the so called **modularization**: the source code is more readable and the debug is easier. For this reason, some compilers, like the ones of GCC, offer options for automatic FI during compilation, in order to let the programmer introduce how many subroutines he/she needs: the functions to be inlined are selected by heuristic rules, such as the function being suitably small. However, such heuristics may not choose the appropriate subroutine, the choice of the programmer should do better.

In most cases, function-call overhead is a negligible fraction of the total run-time of a program. It can become significant only when there are functions which contain relatively few instructions and they account for a substantial fraction of the run-time (as within loops). Inlining is always favorable if there is only one point of invocation of a subroutine. It is also unconditionally better if the invocation of a function requires more instructions (or memory) than moving the body in-line.

```
1 x = cos(v)*(1+sin(u/2))+sin(w)*(1-sin(u/2));
2 t = sin(u/2);
3 x = cos(v)*(1+t)+sin(w)+(1-t);
```

Figure 13: a small piece of C code for calculating the value of a variable `x` in two different ways. In line 1, the CPU must calculate two times the same expression `sin(u/2)` with the need of accessing the content of the variable `u` two times (from a cache line) and invoking two times the function `sin()`and the operator `/`. In line 2, `t` serves as a swap variable, such that its value is used in line 3 for computing the value of `x` with the cost of only one call to `sin()` function. The function-call overhead is avoided in this way.

An efficient design and implementation of a code by a programmer (or scientist) should consider the computational cost of complicated arithmetic operations or mathematical functions, usually implemented as subroutines/functions. This issue is very relevant in the case of Scientific Computing codes: the same mathematical expression or computation through complicated steps involving mathematical functions can be algorithmically implemented in different ways involving operations faster and more efficient than others.

Figure 13 below shows an example of two different codes implementing a C function called `x7()` which calculates *n* values of the 7th integer power of a real number for *n* different arguments `x`. The left code makes use of the `pow()` function of the C standard math library while the right one is based on multiple multiplications. The right code might be faster on any platform and with the code compiled by any compiler (without automatic optimization) because the `pow()` call within the `for` loop takes more time than the execution of four multiplications.

The mathematical functions implemented in standard libraries of compilers are usually coded in such a way in to minimize uniformously the approximation error on the whole domain of the function itself. However, this feature implies an overhead which lowers the performance of the corresponding subroutine/function. When the mathematical function is used in the code on a restrict interval or range, that overhead could be reduced using an alternative mathematical expression instead of the subroutine/function. For example, a truncated series expansion, an approximated formula, the use of a polinomial or rational function fit can be used, taking great care of the error and of its propagation in the expression involving such approximations. Another example involves trigonometric functions: when their implementation are used in iterative loops, their substitution with equivalent algebraic expressions obtained by prostaferesis formulae may result in a speedup of that part of the code.

It should be noted that many compilers offer special optimization flags that activate automatically approximated versions of mathematical subroutines/functions. With such flags or with the intervention of the programmer, any substitution of a standard subroutine/function, implementing a mathematical function with an approximating expression, should be commented and highlighted in the code, with great care to the consequent errors of approximation.

Other types of optimization procedures are exploited by the compiler itself during pre-processing and compilation:

- Figure 14, left column, shows an example of **Dead Code Removal**, i.e. the elimination of a block of source code inside a conditional control structure because the condition will be never satisfied;
- Figure 14, right column, shows an example of **Redundant Core Removal**, i.e. the compiler recognizes automatically that the computing of an arithmetic expression is not necessary because there are unit constants; this optimization cannot be realized when variables are used instead of constants, as reported in Figure 15.

```c
#include <math.h>
void x7(int n, double *q) {
  int i;
  double x, step;
  step = 1.0/n;
  for (i=0,x=0.0;i<n;++i){
    q[i]=pow(x,7.0);
    x += step;
  }
}
```

```c
#include <math.h>
void x7(int n, double *q) {
  int i;
  double x, step;
  step = 1.0/n;
  for (i=0,x=0.0;i<n;++i){
    double x2 = x*x;
    double x4 = x2*x2;
    q[i]=x4*x2*x;
    x += step;
  }
}
```

Figure 14: two different implementations of a C function for computing the values of an integer power in different points of R. The right code is faster than the left one because it makes use of multiplications instead of call to the standard `pow()` C function. Courtesy from G.Amati, F. Massaioli, Optimization and Tuning – 2, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

```
1 b=a+5.0;
2 if (b<a) {
3 ...............
4 }
```

```
1 const c=1.0;
2 ..........
3 f = f*c;
```

Figure 15: left column, an example of C source code with a “Dead Code”, the body of the `if` construct, which will be never executed because the condition is never true; in such case, the compiler recognizes and eliminates from the source code the “dead” part; right column, line 3 contains an update expression for the variable `f` which does not produce any change for it, due to the constant `c=1.0`, so the compiler removes it from the source code. Courtesy from G.Amati, F. Massaioli, Optimization and Tuning – 2, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

The caption of Figure 15 introduces an important issue regarding loop control structures, the **computational costs of iteratives loops**: sometimes the direct explicitation of the expressions within a loop for the different values of the iterative index makes the code more performant. However, this type of implementation, called **loop unrolling**, makes the code less readable, the debugging more difficult and requires more time and hardwork to the programmer. Modern compilers can exploit automatically the advantages of loop unrolling, creating an object code with the unrolled loop, but there are some aspects to be considered before letting the compiler unrolling each loop construct (see Section 4.3).

In computational codes, loop constructs are usually used for accessing different elements of arrays. The more demanding operation within a loop is the mapping from the index of the array element to the corresponding memory location, because an array is a geometrically ordered homogeneous set of values but it is mapped into memory in a sequential way and the CPU accesses the memory sequentially too.

For example, a C array `v[][][]...[]` of dimension `n` (`n` indexes), is mapped into memory according to the rule that the address of the element increases sequentially with the increase of the index most at right respect to a certain index kept fixed. In Fortran, the rule is the opposite, the address increases with the increase of the index most at left. It is said that C arrays are arranged in memory according to rows, while Fortran ones according to columns, because that is what happens in the case of 2D arrays.

Figure 18 shows such two different ways of arranging arrays in memory according to C and Fortran standards, respectively.

```
int i,ip;
for (i=0;i<nfluid;i++) {
  i4=tp[i];
  rho[i4]=0.0;
  u[i4]=0.0;
  v[i4]=0.0;
  w[i4]=0.0;
  for (ip=0;ip<npop;ip++) {
      rho[i4]=rho[i4]+pop[ip,i4];
      u[i4]=u[i4]+pop[ip,i4]*cx[ip];
      v[i4]=v[i4]+pop[ip,i4]*cy[ip];
      w[i4]=w[i4]+pop[ip,i4]*cz[ip];
  }
}
```

Figure 16: example of a C code derived from a Lattice-Boltzmann-like CFD (Computational Fluid Dynamics) code. The variables `cx[]`, `cy[]`, `cz[]` can assume only -1,0,+1 values. In the code they are introduced as arrays and their values depends on the index of molecules types (`ip`). At compile time, the compiler can not substitute the expression `pop[ip,i4]*cx[ip]` with the equivalent (0 or +/-) `pop[ip,i4]`, it needs to use the law that associates the index `ip` with a specific value. The programmer can choose either such an implementation or another one with direct explicitation of `cx[ip]`, `cy[ip]`, `cz[ip]` for each value of `ip`, avoiding the inner `for` loop. Courtesy from G.Amati, F. Massaioli, Optimization and Tuning – 2, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

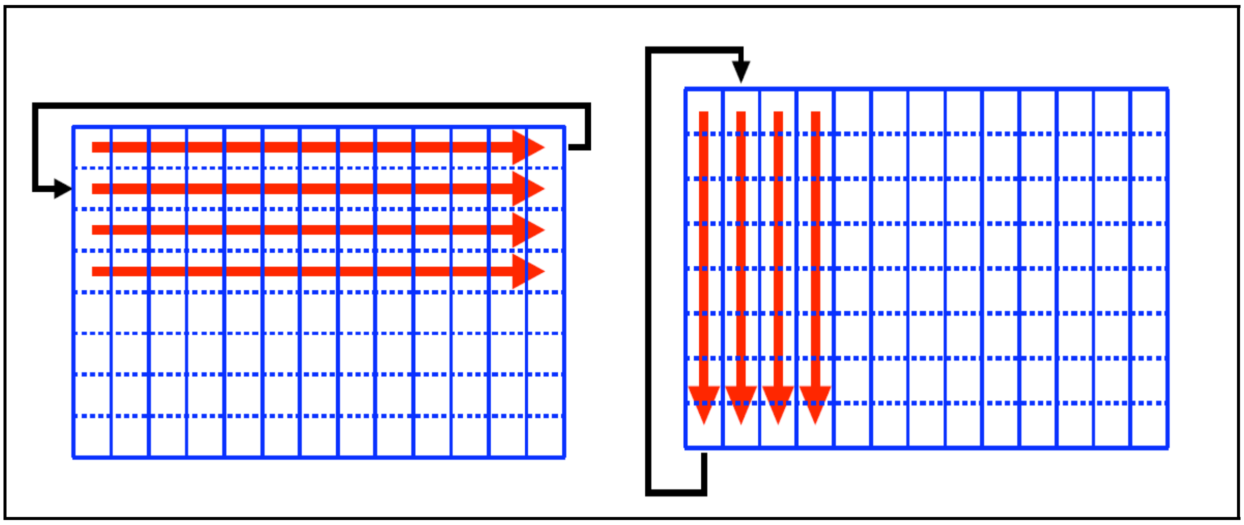


Figure 17: memory arrangement for a 2D array according to C (left column) and Fortran (right) standards. In the left case, elements of the array are mapped into memory (which is accessed sequentially) by rows, i.e. the increment of the index most at right, keeping all the other ones fixed, leads to the access of contiguous elements in memory. In the right case, the sequential access to contiguous array elements can be done incrementing the index most at left, keeping fixed all the other ones (by-columns mapping into memory).

The knowledge of how arrays are mapped into memory is important for the optimization of a loop construct, mainly because modern computers (except for vectorial supercomputers) are based on the hierarchy of memory devices illustrated in Section 3.2.4. In order to take advantage of cache memories and to exploit either spatial or temporal local referencing (or both of them at the same time), nested loops must favour the access to array elements that can better make use of allocation and transfer of data between the different cache and register devices.

A typical example of piece of code whose performance can be greatly improved by the programmer is the one for the computation of the product of two matrices `a` and `b`, implemented in a C code as two 2D arrays `a[][]` and `b[][]`. Figure 18 below shows two different versions of a C code for computing the matrix `c` product of `a` and `b` (`c` = `a`·`b`). The left version does make use of a generic order of the nested loops, while the right one is more performant because it exploits both spatial/temporal local referencing and the properties of data fetching in cache memories.

```
1 for (i=0;i<n1;i++)
2   for (j=0;j<n2;j++)
3     for (k=0;k<n3;k++)
4        c[i][j]=
         c[i][j]+a[i][k]*b[k][j];
```

```
1 for (i=0;i<n1;i++)
2   for (k=0;k<n3;k++)
3     for (j=0;j<n2;j++)
4        c[i][j]=
         c[i][j]+a[i][k]*b[k][j];
```

Figure 18: two different versions of a piece of C code computing the matrix product `c` = `a`·`b`. The left version does not make use of spatial local referencing, because the final rows-columns product at line 4 references elements of the `b` matrix belonging to the same column (with the second index j fixed) but to different rows (changing values fo the first index k). That it means two different accessed elements of the 2D array `b` are not contigous in memory but separated by a stride equal to the number of columns of the `b` matrix itself. The same happens if the order of the nested loops is (`k`,`i`,`j`) or (`k`,`j`,`i`): the first one exploits spatial local referencing for the array `b` but not for array `a`, while it makes use of temporal local referencing for the array `a` because in the last loop over the index `j` the element `a[i][k]` is the same at each iteration; the second configuration is worse than the previous one because it still does not exploit spatial local referencing for the array `a`, makes use of temporal local referencing for the array `b` but at line 4 the final expression accesses different elements of the array `a` which are separated in memory of a stride equal to its number of columns. However, the worst configuration is (`j`,`k`,`i`) because it does not exploit spatial local referencing for the both arrays.

The optimization procedure showed in Figure 18 is called **loop index reordering** and is one of the most important techniques for increasing the speed of a code, making a better use of the memory hierarchy based on different levels of caching data. Modern compilers can implement index reordering during compilation when high level optimization flags are switched on (see Section 4.3), but this automatic optimization is realized only when the nested loops are "regular", i.e. there are are instructions only in the body of the most internal loop. Sometimes, also in the case of regular nested loops, the compiler do not exploit index reordering when the body of one or more loops contains function/subroutine calls or many lines of complicated instructions or when there are many levels of nested loops. For these reasons, the programmer should better implement index reordering in the correct way wherever nested loops are used.

However, index reordering may not lead to better performance of the code in some special cases the programmer should consider with great attention: from Section 3.2.4, it has to be remembered that when an array element is required by the CPU, it is fetched from memory to a cache device together with its neighbours (in terms of memory location) elements. How many elements of the array are fetched into a cache line depends from the structure of the RAM, which is divided into blocks: every block has the same dimension of a cache line. The array element required is fetched into a cache line along with all the other ones present in the same memory blocks. The data type of the array and its dimensions determine how many array elements can be fetched into cache devices at a time. That it means that the spatial locality of array elements depends on the array itself and on RAM and cache dimensions.

In order to try to make a better use of the cache hierarchy, the programmer can use a technique called **cache blocking**, particularly useful with nested loops accessing multi-dimensional arrays elements which cannot be fetched entirely in the cache memories: the nested loops can be subdivided into inner nested loops accessing smaller blocks of the array that can be entirely fetched into cache. Figure 19 below shows an example of calculation (transposing a matrix) that can take advantage of cache blocking.

```
for (i=0;i<n1;i++)
   for (j=0;j<n2;j++)
      a[j][i]=b[i][j];
```

```
for (ii=0;ii<n1;ii+dii)
  for (jj=0;jj<n2;jj+djj)
      for (j=jj;jj<=jj+djj-1;j++)
          a[j][i]=b[i][j];
```

Figure 19: the left column shows an example of C code for the transposition of a matrix (`b`). In this case, the index reordering of the nested loops may not have a good effect on the speed-up first of all because spatial local referencing is exploited on accessing the elements of the matrix `b` but only in the case its second dimension, the number of columns, is not too large and the data type of b does not require too much bytes compared to the capacity of each cache line. At the first iteration of both loops, the fetch of `b[0][0]` in a cache line implies fetching also `b[0][1]`, `b[0][2]`, ..., up to filling all the cache lines. If the previous conditions are not satisfied, it may happen that only the element `b[0][0]`, ..., `b[0][n'2]` are fetched into the cache memory (at one level), with `n'2` < `n2`. Then, the call for `b[0][n'2+1]` implies a cache wipe out and a new cache fetching from RAM memory. However, if the previous conditions may be satisfied not only `b[0][0]`, ..., `b[0][n2]` are fetched into the cache memory at one time but maybe also `b[1][0]`, ..., `b[1][n'1]` with `n'1` < `n1`, which are already ready fo the second iteration of the external loop. These observations underline the importance of knowing the hardware characteristics of the cache memory devices of the computer used. However, it is very time-consuming and hard-working to compare the different cache memories capacity with the dimensions of the array to be treated. The right column shows a different version of the same C code for transposing matrix b into matrix a using the cache blocking technique: using 4 loops instead of 2, with the 2 outermost for limiting the range of the two index i and j, it may be possible that at the first iteration of each loop, the call for the element `b[0][0]` implies fetching into the cache all elements up to `b[0][jj+djj]`. In that way, the innermost loop is run faster because all the elements of `b` needed are still in a cache level. In this case, the asset of the optimization derives from a good choice of the parameter `djj`, the step for making blocks along the rows of `b`. The nested loop might have different performance according distinct values of `djj`: for fixing it it should be useful to know the capacity of lower level cache memory device and the dimension of its cache lines.

There are two other issues connected with cache memory hierarchy which the programmer should take into account in writing an efficient code from the point of view of exploiting the potentialities of caching: two bottlenecks of cache memory functioning should be avoided.

The first bottleneck is called **cache capacity miss** and consists in using only a limited number of cache lines, then reducing the speed of the code because more load instructions must be excuted directly from the RAM memory.

The second bottleneck has been already cited in Section 3.2.4, **cache trashing**: a cache miss occurs because a data is not located in cache memory, so it must be fetched from RAM to a cache line, thus substituting the whole line and loosing previously fetched data which has not yet been used by the CPU and must be fetched subsequently. This process is very time consuming and lower the performance of the code considerably, even worse than in the case of not having the cache hierarchy.

From the programmer point of view, it is very complicated to avoid the two bottlenecks because they are related both to the machine hardware and to the code algorithm and problem dimensions. It is still very difficult to understand if one's code is affected by these two ineffective way of cache functioning. However, there are some possibilities in order to “fight” against these two bottlenecks. In what follows, the reader can find only some suggestions and partial indications because these issues usually are very complicated and pure technical stuff which can not be covered in this small report. For more details, see ....

The first consideration is that L1 and L2/L3 cache devices use different methods for memory locations treatment: L1 devices makes use of logical addresses (refering to the virtual memory, the one that the process sees and is transparent to the programmer) while L2/L3 devices work with physical addresses. This difference implies that the programmer can actually control only how data are fetched into L1 cache, while the performances of L2/L3 memories depend on the physical memory allocated for a process (a running program) and can change from run to run. There are some “tools” at the Operative System level to try to control the use of L2/L3 devices by a program but they are not treated here because they constitute a thorough technical issue.

These difficulties in controlling how cache memories operate are not in contrast with the previous treated “tricks” about index reordering and cache blocking: these methods in writing a source code regards the general way of cache memory fetching and their usefulness remain valid in any case for increasing teh performance of the code. However, capacity miss and cash trashing can still happen also writing a code according to such “rules of thumb”.

While capacity miss is a very tight problem difficult to work out, there is a methodology for facing the shortcomings due to cache trashing called **padding**. Figure 20 shows an ideal case where padding may be very useful in avoiding cache trashing. It is an ideal case because that problem (vectorial sum of two 1D arrays) is actually worked out with high performance using the SSE instructions, inserted in the object code automatically by the compiler when some special options are used (see Section 4.3).

Cache trashing, as cited before and in Section 3.2.4, consists in an unefficient way of fetching subsequently data into a cache device different times, because different data are required: due to cache line fetching, that means loosing data already in a cache line, not yet used but actually needed by the code at a later time, so they must be re-fetched at a later time. Cache trashing is very frequent when two or more data flows make reference to a restricted set of cache lines. It does not increase the number of load/store operations from/to the RAM memory, as one would be tempted to think of, but actually it does raise the “traffic” on the FSB (Frontal Side Bus), the bus that connect the RAM to the cach hierarchy. Cache trashing is almost present when data flows have a stride (“distance” between two required data at two different steps of the flow) which is a power of 2.

Figure 21a and 21b show an example of data flow which has a pattern of access to the cache with a stride equal to 4 blocks (of memory, i.e. 4 cache lines). In this case, padding consists in introducing into the source code dummy variables that break down this pattern of data allocation, avoiding the need of updating a cache line with new data before having used all the previous data in that line.

It is very difficult to identify cache trashing from the code profiling. Sometimes it is useful to try to test the run of a code with different dimensions of the problem to be solved (e.g. different global dimensions of the arrays used). When the dimension of the problem is a power of 2, the code may slow down significantly. However, this approach to cache trashing implies high computational costs: different run are needed with different dimenions of the problem. The trade-off between speed-up of the code and consume of computing time at diposal must be taken in great consideration.

To avoid this approach to overcome cache trashing, many CPU vendors offers specific programs that can analyze and realize a statistics of measured events by some CPU built-in registers. For example, in many CPU there are registers that count the total number of clock cycles needed for the execution of each block of code. They have been added to CPU hardware in order to debug the CPU way of operation itself. Other types of registers dsigned for the same aims collect data about how many load/store operations are executed, how many cache misses happen and so on. Every CPU vendor has implemented different types of registers for tracking different variables and the corresponding proprietary software produce different types of statical analyses.

It should be useful, however, to cite the existence of some multi-architecture open source tools that can make many of these measurements during code run. One of the most famous is the PAPI project and software, developed at the Innovative Computing Laboratory, University of Tennessee. PAPI stands for Performance API and provides a consistent interface to performance hardware counters. It is available for most architectures and implementations of CPUs and most of Oss and substantially provides two main APIs (Application Programming Interfaces) to access the underlying counters hardware: a low level interface manages hardware events in user defined groups called *EventSets*; a high level interface provides the ability to start, stop and read the counters for a specified list of events. PAPI is not specificaly aimed at parallel programs. Another project at the Innovative Computing Laboratory, University of Tennessee is dedicated to the development of a software tool for automatic performance analysis of parallel codes, written in MPI 1, MPI 2.0, OpenMP ans SHMEM one-sided communication (look at the KOJAK project).

Coming back to iterative loops, there are other important issues about improvement of the performance of a code still at the source code level:

- both in C and in Fortran, the loop index must be an integer data, otherwise the compiler must transform it in an integer value with consequent overhead and problems of approximations (conversion from a floating point data to an integer by rounding);
- many automatic optimizations by the compiler presented above are not realized in special conditions, such as when a loop's body is too large or there are too many conditional control constructs; in such cases, the programmer must implement them by his/her own;
- frequent use of the CSE technique introduces an overhead due to the use of too many swap variables.

```
for (i=0;i<n;i++)
  c[i]=a[i]+b[i];
```

Iteration `i=0`:

1. search for `a[0]` in L1 cache -> cache miss
2. load `a[0]` from RAM;
3. cache line fetching: copy of `a[0]`, ..., `a[n-1]` into a cache line;
4. copy of `a[0]` in a CPU register;
5. search for `b[0]` in L1 cache -> cache miss
6. load `b[0]` from RAM;
7. cache line fetching: copy of `b[0]`, ..., `b[n-1]` into a cache line;
8. copy of `b[0]` in a CPU register;
9. execution of the operation `c[0]=a[0]+b[0]` and storing its result in a CPU register

Iteration `i=1`:

1. search for `a[1]` in L1 cache -> **cache hit** !
2. copy of `a[1]` in a CPU register;
3. search for `b[1]` in L1 cache -> **cache hit** !
4. copy of `b[1]` in a CPU register;
5. execution of the operation `c[1]=a[1]+b[1]` and storing its result in a CPU register

Iteration `i=2`:

....

Iteration `i=0`:

1. search for `a[0]` in L1 cache -> cache miss
2. load `a[0]` from RAM memory
3. cache line fetching: copy of `a[0]`, ..., `a[n-1]` into a cache line of L1;
4. copy of `a[0]` into a CPU register;
5. search for `b[0]` in L1 cache -> cache miss
6. load `b[0]` from RAM;
7. **wiping out the cache line of L1 containing `a[0]`, ..., `a[n-1]`**;
8. **cache line fetching: copy of `b[0]`, ..., `b[n-1]` into the same cache line**;
9. copy of `b[0]` into a register;
10. execution of the operation `c[0]=a[0]+b[0]` and storing its result in a CPU register

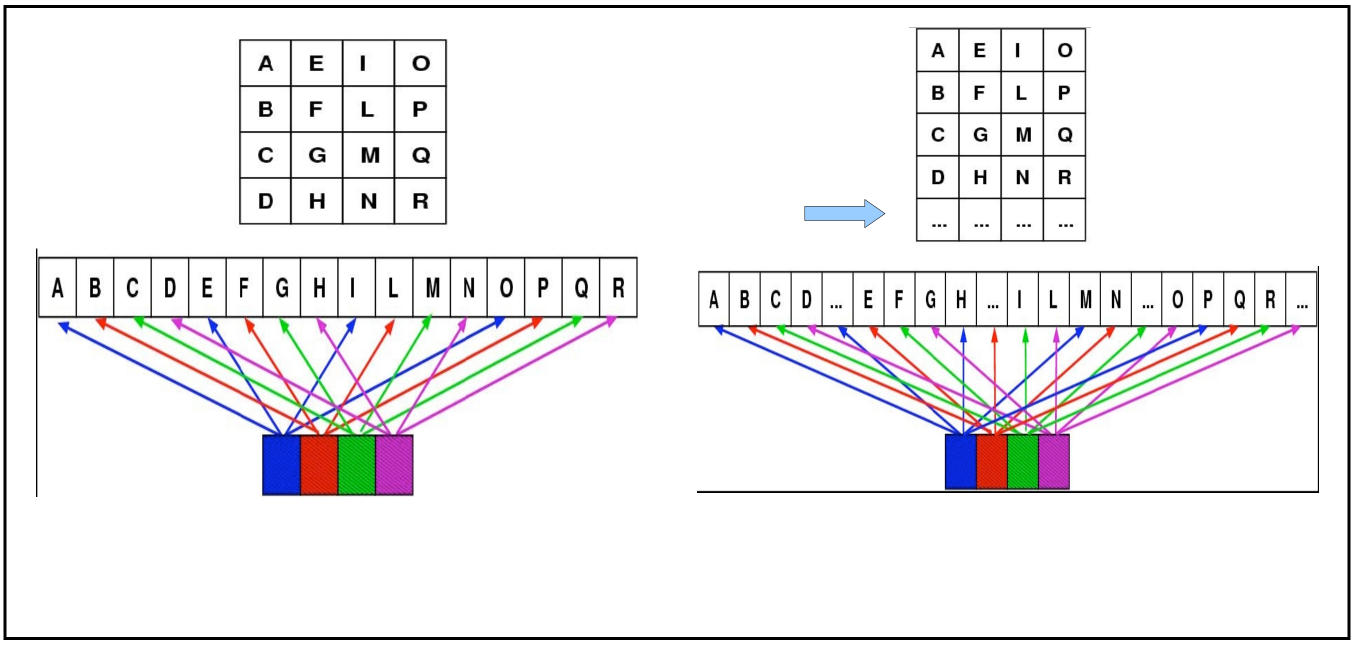
Iteration `i=1`:

1. search for `a[1]` in L1 cache -> cache miss
2. load `a[1]` from RAM memory;
3. **wiping out the cache line of L1 containing `b[0]`, ..., `b[n-1]`**;
4. **cache line fetching: copy of `a[0]`, ..., `a[n-1]` into a cache line of L1**;
5. copy of `a[1]` into a CPU register;
6. search for `b[1]` in L1 cache -> cache miss
7. load `b[1]` from RAM;
8. **wiping out the cache line of L1 containing `a[0]`, ..., `a[n-1]`**;
9. **cache line fetching: copy of `b[0]`, ..., `b[n-1]` into the same cache line**;
10. copy of `b[1]` into a register;
11. execution of the operation `c[1]=a[1]+b[1]` and storing its result in a CPU register

Iteration `i=2`:

....

Figure 20: a simple C code for the vectorial sum of two 1D arrays a and b is presented along with two possible pseudo-code versions of the flow of operations when that block of code is run. The length n of the two arrays and their data type are assumed such that a cache line fetching results in fetching all the elements of the array whose element is required. The first pseudo-code refers to the case without cache trashing, for example because the a[0] and b[0] addresses refer to distinct location of RAM memory which corespond to diferent cache lines. Then, after the first iteration, when searching for a[1] and b[1] into L1 cache, two cache hit occur because those data are already stored in distinct cache lines. In this way, the cache memory principle is used at its best. In the second pseudo code, a cache trashing occurs, e.g. because the a and b arrays are located into regions of RAM memory mapped into the same cache line and, although a L1 cache device is usually 4/8 way associative, the corresponding cache lines of diferent slots are already occupied by other data belonging to other parts of the code. It can be seen that at step 7 of the first loop iteration the cache line must be voided before fetching the b array (step 8). Then, at the second iteration of the loop, a cache miss occurs also on the quest for a[1] because at the previous step the cache line have been fetched with b[] elements. At step 3 of this second iteration, the cache line must be voided again, re-fetched with a[] elements and the same happens subsequently when b[] elements are invocated. This unfect/fetch from RAM to L1 cache device occurs at each iteration and constitutes an overhead for the execution of the code. In this case, not only the process goes on as if the cache memory hierarchy does not exist but its performance is even worse: a direct fetching from the RAM will make the code faster and more perfomant. This example has been extracted from G.Amati, F. Massaioli, Optimization and Tuning – 3, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.



A E I O
B F L P
C G M Q
D H N R
A B C D E F G H I L M N O P Q R
A E I O
B F L P
C G M Q
D H N R
... ...
A B C D ... E F G H ... I L M N ... O P Q R ...

Figure 21: a) left column, example of cashe trashing in data fetching. b) right column, example of how padding avoids cache trashing. In the left column case, the pattern of access to data block is A -> E -> I -> O so that after having fetched in the blue cache line the A block, the quest for a data in E requires a re-fetching of the same cache lines. In this case it should be noted that 4 blocks of memory (A,B,C,D, which can be 4 different elements of a Fortran array) the whole cache device in this schematic example. Addressing subsequent elements in the array, e.g. E, F, G, ..., means unfetch and re-fetch the cache. In the right column, it is shown how padding is exploited: the 2D array is increased, by the programmer, of one “dummy” row, i.e. four elements, one for each column. From the arrangement in RAM memory point of view, that means a shift of one memory block forward, that is forward shift also in the cache line mapping. When an element occupying the E block is required, it has to be fetched into cache from RAM, substituting the B data, but the cache line containing A data is not wiped out. The cartoons are courtesy taken from G.Amati, F. Massaioli, Optimization and Tuning – 3, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

4.3 Assembly-code level optimization by compilers

As seen in the previous Section, many changes in the source code can be automatically implemented by the compiler if some specific options are passed to the compiler itself when it is invoked. However, it has been shown in different cases and for different types of optimization techniques that it is better the programmer holds control of its code implementation where he/she can because the compiler may not do a useful change.

There are other types of optimization procedures that either do not/can not involve the source code directly because they are strictly connected to hardware details very difficult to be taken into account by the programmer (also for the hacker one) or are easily implementable by the compiler, avoiding to the programmer hardwork. These optimization tricks try to exploit the CPU built-in parallelism due to the use of instructions look ahead, pipelining, multiple functions units and SIMD instructions (or vectorial instructions), discussed in Sections 3.2.6-9.

In order to exploit such an intrinsic parallelism, many problems must be solved before and during runtime:

- the pipelines of the different functional units must be feeded continuously in order to execute more instructions per unit of time;
- the strict sequentiality of the code must be broken in order to feed the different pipelines, many instructions and data flows must be created;
- how to work out the relationships between different instructions ?
- how to coordinate the elaboration on data among different instructions flows ?

Some of these issues are of competence of the CPU and are tackled only at runtime by specific hardware units that implement the instructions look-ahead, also called **Out Of Order Execution** (**OOOE**), and the **branch prediction**.

The OOOE technique consists in a dynamic reordering of the instructions, as derived from the machine code, at runtime: the original sequence of operations as imposed by the programmer with his/her coding is completely broken, the CPU may start to execute an instruction which comes after another in the original assembly code if that instruction has already at disposal its operands; in other cases, the CPU may decide to postpone the execution of an operation at later times and to go on along the flow of the code because the needed data are not still ready. For such changes at runtime, the CPU makes intesive use of such techniques as renaming of registers, aggregation of memory load and store instructions and branch prediction.

Branch prediction consists in avoiding the execution of the control instructions of an `if` or loop construct, guessing the result of such a control through a random choice and going on along the flow of the code. This may improve the performance of a run because when the CPU arrives at the conditional construct the data necessary for the control of the condition may not be at disposal, so an interruption in the execution of instructions by some units should happen. With branch prediction, the CPU can maintain at full work each of its units. When the conditional expression is then controlled, the CPU compares the result of the control with its forecast: if the prediction has had success, the CPU has managed not to break some instruction flow, if not the CPU must come back to the point of the conditional expression and execute the other instructions.

Branch prediction is operated by the CPU with the help of a specific unit called **Branch Prediction Unit** which is dedicated to the collection of data about the most recently results (during runtime) of branching constructs. On the basis of such collections of data, the CPU tries to forecast the result of a new conditional expression still to be evaluated. This approach to tackle the issue of branching in the flows of a program may be very effective when the code contains a significant number of conditional constructs with similar patterns (e.g. loops or `if` statements for managing errors in I/O operations).

`If-else` constructs are treated by the CPU at runtime also with another approach called predicative: both blocks of codes are executed although the value of the conditional expression has not yet been calculated; when this calculation has been done, only one flow is brought on. This technique may be very performant only in the case of `if-else` constructs with small and homogeneous bodies and let the program use in a more effective way the pipelining because different pipes are feeded with different instructions and data flows.

In order to help the CPU with OOOE, branch prediction and exploitation of all resources connected with pipelining, the code can be written according to certain approaches, instrumented with additional instructions or changed by the compiler.

The **FI** (**Function Inling**), treated in the previous Section, is an example of technique that the programmer should adopt at the source code level in order to decrease the flow interrupts within the code: as reported previously, each function/subroutine call introduces an overhead, the CPU executes a jump to another portion of code, data may be copied from memory locations to other ones if parameters must be passed to the function/subroutine, thus the pipeline gets affected by interrupts or "bubbles" (some stages of a pipeline remain void for many clock cycles). GCC C/C++ compilers (`gcc` and `g++`) offer specific compilation flags to be switched on in order to automatically perform function inlining. Table 2 below contains a summary of such flags and what they instruct the compiler to do. GNU C provides several language features not found in ISO standard C, among which the possibility of adding into the source code keywords that makes the compiler implement function inling. The keyword `inline` must be inserted in the line declaring the function, before the definition of the returned data type. As stated in the GCC manual "By declaring a function `inline`, you can direct GCC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions is an optimization and it really "works" only in optimizing compilation." [GCC Manual, Ver. 4.4.1]

compiler flag	action
`-finline-functions`	Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.
`-finline-functions-called-once`	Consider all `static` functions called once for inlining into their caller even if they are not marked `inline`. If a call to a given function is integrated, then the function is not output as assembler code in

	its own right. Enabled if `-funit-at-a-time` flag is switched on too.
`-fearly-inlining`	Inline functions marked by `always_inline` and functions whose body seems smaller than the function call overhead early before doing `-fprofile-generate` instrumentation and real inlining pass. Doing so makes profiling significantly cheaper and usually inlining faster on programs having large chains of nested wrapper functions. Enabled by default.
`-finline-limit=`*n*	By default, GCC limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as `inline` (i.e., marked with the `inline` keyword or defined within the class definition in c++). *n* is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of *n* is 600. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code will be inlined (which presumably means slower programs). This option is particularly useful for programs that use inlining heavily such as those based on recursive templates with C++. Inlining is actually controlled by a number of parameters, which may be specified individually by using '`--param name=value`'. The `-finline-limit=`*n* option sets some of these parameters as follows: `max-inline-insns-single` is set to *n*/2. `max-inline-insns-auto` is set to *n*/2. `Min-inline-insns` is set to 130 or *n*/4, whichever is smaller. `max-inline-insns-rtl` is set to *n*.
`-fkeep-inline-functions`	In C, emits `static` functions that are declared `inline` into the object file, even if the function has been inlined into all of its callers. This switch does not affect functions using the `extern inline` extension in GNU C. In C++, emit any and all inline functions into the object file.

Table 2: list of GCC gcc/g++ options for automatic function inling optimization.

Another optimization technique that could be implemented both by the programmer at source code level and by the compiler is **loop unrolling**, introduced in Section 4.2. Figure 22 shows an example of piece of C code with a nested loops construct which can benefit of unrolling. In the original version of the coded loop (left column), it can be seen that there are two basic arithmetic operations for each iteration (an addition and a multiplication) and two basic interdependent instructions (the array element `c[i][j]` is updated after the multiplication has been done). The overhead associated to this construct is associated to 1) the updating of the loop indexes after each iteration 2) the mapping between an indexed array element and the corresponding memory location 3) the evaluation of a conditional expression at the end of each internal loop. The right column shows an equivalent coding of the same construct with more performant than the previous one: the body of the innermost loop contains two independent couples of inter-dependent instructions, which can be assigned to two different instructions pipelines; the total number of iterations over all loops has been halved, so the number of evaluations of conditional expressions; two data streams have been created but both of them can have direct access from the cache to the needed data, due to a correct use of index ordering; at leat one variable is reused between the two couples of instructions.

The GCC compilers offers special compilation options to turn on automatic loop unrolling optimization procedures. The two basic loop unrolling flags are `-funroll-loops` and `-funroll-all-loops`: the first one makes the compiler to unroll each loop whose total number of iterations can be determined at compile time or upon entry to the loop; the second one makes the compiler unroll every loop even if the total number of iteration is uncertain when the loop in entered at runtime. Both optimization options lead to a larger code, but the performance might increase. It is better not to abuse of such options because the performance gained by loop automatic unrolling is case-by-case dependent, so it should be better for the programmer choosing where to apply loop unrolling directly in the source code.

```
for (i=0;i<n1;i++)
  for (k=0;k<n3;k++)
     for (j=0;j<n2;j++)
        c[i][j]=
         c[i][j]+a[i][k]*b[k][j];
```

```
for (i=0;i<n1;i=i+2)
  for (k=0;k<n3;k++)
     for (j=0;j<n2;j++) {
       c[i][j]=
        c[i][j]+a[i][k]*b[k][j];
       c[i+1][j]=
        c[i+1][j]+a[i+1][k]*b[k][j];
      }
```

Figure 22: a piece of C code for the calculation of a matrix product with the optimized index ordering as proposed in Figure 18. The left column contains the original implementation of the nested loop construct, while the right one a modified version with an external loop on the index `i` with a doubled step: two instructions in the body of the innermost loop calculates two elements of the resulting product matrix insted of one at each iteration. This modification is an example of loop unrolling (iteration on indexes are substitued by direct explicattion of instructions on array elements). The loop unrolling technique can improve the performance of the code for many reasons: better use of cached data, creation of many distinct instructions and data streams that can be pipelined in different "channels"; reduction of the number of iteration, i.e. of controls of conditional expression at the end of loops (so reduction of jump instructions). However, loop unrolling should not be considered as a "magic rule" for getting more performance: it should be tested and adopted on a case-by-case, trying diffent "depth" levels of unrolling. The example is derived from G.Amati, F. Massaioli, Optimization and Tuning – 2, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

Loop merging and **loop splitting** are other two techniques involving loop constructs which should be considered when trying to optimize a code but they are not covered here for brevity.

GCC compilers have general compiler options that can switch on many automatic optimization procedures together. There is a hierarchy of such options, all of them are defined as `-Ox` where `x` is a number between 0 and 3 or the letter "`s`". Table 3 shows the list of such compiling options and which types of optimizations they instruct the compiler to apply. As a general suggestion, it should be better not to abuse of these general collected optimization flags. It should be also reminded that the higher the level of optimization `x` the higher the number of changes within the code: at different stages of code development and deployment different levels should be used. For example, `-O0` must be used doing debugging; `-O1` or `-O2` during the study of the performance of the code or its profiling; `-O3` is very dangerous because the structure of the code may be completed modified, even certain particular constructs chosen by the programmer for the optimization itself; it derives that sometimes the performance with `-O3` flag turned on can be worse than without any `-Ox` flag.

Optimization compiler option	What it performs on code
`-O0`	The compiler does not perform any type of optimization.
`-O -O1`	The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. It turns on a set of compilation options, among which `-floop-optimize` (which performs loop optimizations such as moving constant expressions out of loops, simplifying exit test conditions and optionally doing strength-reduction as well), `-fbranch-guess-probability` (GCC will use heuristics to guess branch probabilities if they are not provided by profiling feedback, heuristics based on the control flow graph), `-fdelayed-branch` (which attempts to reorder instructions to exploit instruction slots available after delayed branch instructions), `-fif-conversion` (which attempts to transform conditional jumps into branch-less equivalents, including the use of conditional moves, min, max, set flags and abs instructions, and some tricks doable by standard arithmetics) and `-fif-conversion2` (which does use conditional execution (where available) to transform conditional jumps into branch-less equivalents), `-ftree-dce` (which performs dead code elimination (DCE) on trees), `-fmerge-constants` (which attempts to merge identical constants, e.g. string constants and floating point constants, across compilation units).
`-O2`	The compiler performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining in this case. As compared to `-O`, this option increases both compilation time and the performance of the generated code. `-O2` turns on all compilation flags of `-O` plus others among which: -fthread-jumps, which performs optimizations where we check to see if a jump branches to a location where another comparison subsumed by the first is found; if so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false; `-fcrossjumping`, which performs cross-jumping, a transformation that unifies equivalent code and save code size, although the resulting code may or may not perform better than without cross-jumping; `-fgcse`, which performs a global common subexpression elimination pass; this pass also performs global constant and copy propagation; `-fcse-follow-jumps`, which, in common subexpression elimination, scans through jump instructions when the target of the jump is not reached by any other path; for example, when CSE encounters an `if` statement with an `else` clause, CSE will follow the jump when the condition tested is false; `-fcse-skip-blocks`, which is similar to

	-fcse-follow-jumps but causes CSE to follow jumps which conditionally skip over blocks; when CSE encounters a simple if statement with no else clause, -fcse-skip-blocks causes CSE to follow the jump around the body of the if; -fstrength-reduce, which performs the optimizations of loop strength reduction and elimination of iteration variables; -fschedule-insns, that attempts to reorder instructions to eliminate execution stalls due to required data being unavailable; this helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required (exploittion of OOOE at runtime); -fschedule-insns2, similar to -fschedule-insns but requests an additional pass of instruction scheduling after register allocation has been done; this is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle; -falign-functions=*n*, where n is a machine dependent parameter, aligns the start of functions to the next power-of-two greater than *n*, skipping up to *n* bytes; for instance, -falign-functions=32 aligns functions to the next 32-byte boundary, but -falign-functions=24 would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less; -falign-jumps is equivalent to -falign-jumps=*n,* with *n* as a machine dependent parameter, and aligns branch targets to a power-of-two boundary, for branch targets where the targets can only be reached by jumping, skipping up to *n* bytes like -falign-functions; -funit-at-a-time parses the whole compilation unit before starting to produce code, allowing some extra optimizations to take place but consuming more memory (in general). There are some compatibility issues with unit-at-at-time mode: • enabling unit-at-a-time mode may change the order in which functions, variables, and top-level asm statements are emitted, and will likely break code relying on some particular ordering. The majority of such top-level asm statements, though, can be replaced by section attributes. • unit-at-a-time mode removes unreferenced static variables and functions. This may result in undefined references when an asm statement refers directly to variables or functions that are otherwise unused. In that case either the variable/function shall be listed as an operand of the asm statement operand or, in the case of top-level asm statements the attribute used shall be used on the declaration. • Static functions now can use non-standard passing conventions that may break asm statements calling functions directly. Again, attribute used will prevent this behavior.
-O3	Optimize yet more. -O3 turns on all optimizations specified by -O2 plus the -finline-functions, -funswitch-loops (moving branches with loop

	invariant conditions out of the loop, with duplicates of the loop on both branches, modified according to result of the condition) and `-fgcse-after-reload` (a redundant load elimination pass is performed after reload with the purpose to cleanup redundant spilling) options.
`-Os`	Optimize for size. `-Os` enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

Table 3: list of GCC gcc/g++ options for automatic optimization. These optimization flags are general ones which include in only one string of characters, `-Ox` with `x` a integer number from 0 to 3 or the letter `s`, the call to many optimization options. Four different levels o automatic optimizations are determined, while `-Os` is the flag for the optimization dedicated to the memory space occupied by the program.

The `-Ox` (for any `x` value) optimization flags set does not include any kind of option to instruct the compiler to use SIMD extensions in floating points arithmetic instructions. As discussed in Section 3.2.9, the programmer may exploit the power of MMX/SSE/SSE2/SSE3 extensions to the Assembly instruction set either including intrinsics into the code (or directly Assembly language instructions) or relying on compiler special options dedicated to insert SIMD extensions instruction into the Assembly code.

GCC C/C++ compiler has a general option that lets the user choose which type of floating point arithmetics model to be used: `-mfpmath=`*`Unit`*`.` *`Unit`* is keyword which can assume this values:

- `387`, i.e. use of the standard 387 floating point coprocessor present in the majority of chips and emulated otherwise; code compiled with this option will run almost everywhere; the temporary results are computed in 80bit precision instead of precision specified by the type of architecture (see below the `-march=cpu-type` option) resulting in slightly different results compared to most of other chips;
- `sse`, i.e. use of scalar floating point instructions present in the SSE instruction set; this instruction set is supported by Pentium III and newer chips, in the AMD line, by Athlon-4, Athlon-xp and Athlon-mp chips; the earlier version of SSE instruction set supports only single precision arithmetics, thus the double and extended precision arithmetics is still done using 387; later version, present only in Pentium IV and the future AMD x86_64 chips, supports double precision arithmetics too; for the i386 compiler, you need to use `-march=cpu-type` (see below) plus `-msse` or `-msse2` switches to enable SSE extensions and make this option effective; for the x86_64 compiler, these extensions are enabled by default; the resulting code should be considerably faster in the majority of cases and avoid the numerical instability problems of 387 code, but may break some existing code that expects temporaries to be 80bit;
- `sse,387`, i.e. an attempt to utilize both instruction sets at once; this effectively double the amount of available registers and on chips with separate execution units for 387 and SSE the execution resources too; this option must be used with care, as it is still experimental, because the GCC register allocator does not model separate functional units well resulting in instable performance.

The `-march=cpu-type` is a more general option to the compiler that implies automatically the `-mtune=cpu-type` one. This last option, turned on, instructs the compiler to apply every possible optimization procedure in order to adapt the code to the specified implementation of CPU indicated by the argument `cpu-type`. The number of CPU implementations supported by the last (at the time this report has been written) release of GCC (version 4.1.1) is cospicous: i386 (original Intel i386 CPU); i486 (Intel 486); i586 or pentium (Pentium without MMX); pentium-mmx (Intel Pentium core with MMX support); i686 or pentiumpro (Intel PentiumPro); pentium2 (Intel Pentium II based on Intel PentiumPro core with MMX instruction set support); pentium3 or pentium3m (Intel Pentium III base on PentiumPro core plus MMX and SSE instruction sets support); pentium-m (low power version of Pentium III with MMX, SSE and SSE2 instruction sets support); pentium4 or pentium4m (Intel Pentium IV with MMX,SSE and SSE2 instruction sets support); prescott (improved version of Intel Pentium IV with MMX, SSE, SSE2 and SSE3 instruction sets support); nocona (improved version of Intel Pentium IV with 64-bit extensions, MMX, SSE, SSE2, SSE3 instruction sets support); k6 (AMD K6 CPU with MMX instruction set support); k6-2 or k6-3 (improved version of AMD K6 with MMX and 3DNOW! Instruction sets support); athlon or athlon-tbird (AMD Athlon CPU with MMX, 3DNOW! And enhanced 3DNOW! Instructions sets support); athlon-4 or athlon-xp or athlon-mp (improved AMD Athlon CPU, with MMX, 3DNOW!, enhanced 3DNOW! and full

SSE instruction sets support); k8 or opteron or athlon64 or athlon-fx (AMD K8 core based CPUs with x86_64 instruction set support, which supersets MMX, 3DNOW!, enhanced 3DNOW! and 64-bit instruction sets extensions). These are the most diffuse x86/x86_64 CPU implementations. For other types of architectures (IBM RS/6000, IBM PowerPC, SPARC, MIPS, Dec Alpha, VAX, zSeries, IA-64, etc.) there are other specific tuning options [Manuale GCC].

A particular difference between `-march=cpu-type` and `-mtune=cpu-type` options should be remarked: while the first one implies the second one and let the code be run only on the specific CPU implementation indicated by the parameter `cpu-type`, the second one is less restrictive and optimizes the code for the indicated CPU but let it be usable on old i386 platforms.

Finally, there are other generic options that instruct the compiler to use SIMD extensions of the instruction set not only in the restricted case of floating point arithmetics instructions: `-mmmx`, `-msse`, `-msse2`, `-msse3`, `-m3dnow`. These options will let GCC compilers to use the instructions of the corresponding extended instruction sets even without using `-mfpmath=sse` option. It should be remembered that GCC compilers offers to the programmers built-in functions that implement at the high level the Assembly instructions of the extended instruction sets [Stallman2006].

4.4 The Intel Fortran-C/C++ Linux compilers for x86/x86_64 architectures as an example of optimal compiler for Intel CPUs

GCC compilers are the most used ones within the Linux communities, also in the HPC ones. GNU/Linux Oss and programs running over them are usually compiled by GCC tools; the kernel itself of a GNU/Linux OS is compiled with such tools.

However, the Intel Fortran-C/C++ compilers for GNU/Linux platforms have been getting more space, particularly in HPC, first of all because Intel offers them free with a special non-commercial license for a single seat. For Academic or commercial use, they are not free, along with the same compilers for MS Windos platforms.

Intel has got an interest in developing compilers by its own in order to demonstrate and provide full access to the whole range of capabilities provided by its CPUs. The considerations in this Section just refer to the C/C++ compilers but they are applicable to the Fortran compiler as well.
The last release of Intel C/C++ compilers for Linux platform is the 9.1 and can be found at the URL http://www.intel.com/software/products/compilers/clin . First of all, like GCC, Intel adheres to the latest ANSI C/C++ standards such as C99 for the C compiler. Second of all, Intel C/C++ compilers supports most of GCC C/C++ languages extensions and the OpenMP 2.5 standard for Multi-Threading parallel programming on shared-memory parallel computers (see Section 5.1.2.1). Intel C/C++ compilers are also **binary compatibles** to GCC ones. This is an important feature: to be able to mix object codes created by arbitrary distinct compilers, most manufacturers tend to use a common **A**pplication **B**inary **I**nterface (**ABI**), a set of common rules to describe object files. Object files can only be interlinked if they are ABI-compatible. Intel C/C++ 9.1 compilers for Linux are binary or object-file compatible with C-language binary files created with GNU gcc and C++ binary compatible with g++ versions 3.2, 3.3, 3.4, and 4.0. So, it is possible to create programs using, e.g., static or dynamic libraries already compiled with GCC tools.

Beyond the high level GCC compatibility, what makes Intel compilers very attractive for the HPC user (but not only) is the fact that they are tuned for Intel CPU implementations, both x86, x86_64, EMT64 (Itanium) and Multi-Core families. They offer 6 main types of optimization options and tools which are fine tuned for Intel CPU architectures:

- **InterProcedural Optimization** (**IPO**), which dramatically improves performance of small- or medium-sized functions that are used frequently, especially programs that contain calls within loops;
- **Profile-Guided Optimization** (**PGO**), that improves application performance by reducing instruction-cache thrashing, reorganizing code layout, shrinking code size, and reducing branch mispredictions;
- an **Automatic Vectorizer** which parallelizes code and aligns data, including loop peeling, to generate aligned loads and loop unrolling to match the prefetch of a full cache line;
- **High Level Optimization** (**HPO**) that delivers aggressive optimization with loop transformation and pre-fetching;
- optimized code debugging with the Intel® Debugger, which is one of the tools coming with the Intel C/C++ compilers package, very useful for improving the efficiency of the debugging process on code that has been optimized for Intel® architecture;
- Multi-Threaded Application Support, including OpenMP and auto-parallelization for simple and

efficient software threading.

The IPO consists in a set of techniques that uses multiple files or all the source files composing a program to detect and perform certain types of optimization actions. Usually, compilers compiles each source code file separately and applies optimization procedures to each of them separately too. When a program is built using IPO, each source code file is compiled using a specific option (`-ip` or `-ipo`, interprocedural optimization for single files or across file boundaries). The corresponding object code contains special information written with an **Intermediate Language** (**IL**) used by the compiler, which combines all this information at linking time and analyzes it for optimization opportunities. Typical optimizations made as part of the IPO process include procedure inlining and re-ordering, eliminating dead (unreachable) code, and constant propagation, or the substitution of known values for constants. IPO enables more aggressive optimization than what is available at the intra-procedural level, since the added context of multiple procedures makes those more-aggressive optimizations safe. IPO is available also with GCC compilers but in the case of Intel's one is not restricted to file boundaries.

The Profile-Guided Optimization (PGO) compilation process enables the Intel C/C++ compiler to take better advantage of the processor microarchitecture, more effectively use instruction paging and cache memory, and make better branch predictions. It improves application performance by reorganizing code layout to reduce instruction-cache thrashing, shrinking code size and reducing branch mispredictions.

PGO is a three-stage process, as shown in Figure 23. Those steps include 1) a compile of the application with instrumentation added, 2) a profile-generation phase, where the application is executed and monitored, and 3) a recompile where the data collected during the first run aids optimization. A description of several code size influencing profile-guided optimizations follows:

- basic block and function ordering, i.e. place frequently-executed blocks and functions together to take advantage of instruction-cache locality;
- aid inlining decisions, i.e. inline frequently-executed functions so the increase in code size is paid in areas of highest performance impact;
- aid vectorization decisions, i.e. vectorize high trip count and frequently-executed loops so the increase in code size is mitigated by the increase in performance.

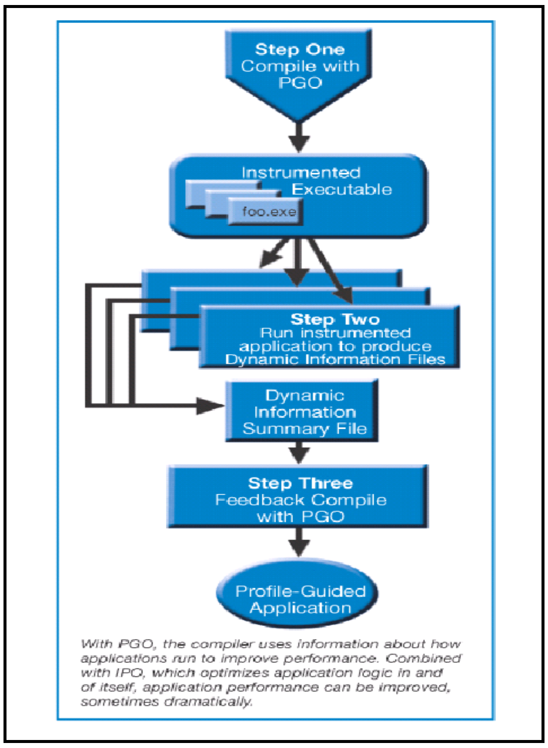

Step One
Compile with
PGO
Instrumented
Executable
foo.exe
Step Two
Run instrumented
application to produce
Dynamic Information Files
Dynamic
Information
Summary File
Step Three
Feedback Compile
with PGO
Profile-Guided
Application
With PGO, the compiler uses information about how applications run to improve performance. Combined with IPO, which optimizes application logic in and of itself, application performance can be improved, sometimes dramatically.

Figure 23: schematic view of the PGO flow subdivided into three different steps, building an instrumented executable code, collecting information during its run in Dynamic Information Files, re-compiling it with the feedback of a summary of data obtained from its instrumented execution. Courtesy from Intel Linux C/C++ 9.1 compiler web page, http://www3.intel.com/cd/software/products/asmo-na/eng/compilers/284264.htm

The Automatic Vectorizer changes the structure of the code to allow machine programs to leverage SIMD extensions. It parallelizes code to maximize underlying processor capabilities. This advanced optimization analyzes loops and determines when it is safe and effective to execute several iterations of the loop in parallel by utilizing MMX™, SSE, SSE2, and SSE3 instructions. Features include support for advanced, dynamic data alignment strategies, including loop peeling to generate aligned loads and loop unrolling to match the prefetch of a full cache line.

HLO (High Level Optimization) is a feature present in the last versions of Intel C/C++ compilers for Linux. It enhances even more the exploitation of two types of optimizations:

- **data prefetching**, used to hide memory access latency, significantly improving performance in many compute-intensive applications, it inserts prefetch instructions for selected data references at specific points in the program, so referenced data items are moved as close to the processor as possible (put in cache memory) before the data items are actually used;
- **loop unrolling**.

The Intel® Debugger enables optimized code debugging (i.e., debugging code that has been significantly transformed for optimal execution on a specific hardware architecture), while usual debuggers can not provide reliable information when debugging an optimized code, because it may have been changed radically from its source code version. The Intel compilers produce standards-compliant debug information for optimized code debugging that is available to all debuggers that support Intel compilers. The Intel Debugger supports multi-core architectures by enabling debugging of multi-threaded applications, providing the following related capabilities:

- an all-stop/all-go execution model (i.e., all threads are stopped when one is stopped, and all threads are resumed when one is resumed);
- list all created threads;
- switch focus between threads;
- examine detailed thread state;
- set breakpoints (including all stop, trace and watch variations) and display a back-trace of the stack for all threads or for a subset of threads;
- a built-in GUI that provides a thread panel (on the Current Source pane) that activates when a thread is created, and that allows an operator to select thread focus and display related details.

Intel Debugger program (`idb`) can be invoked with the `-gdb` option for letting it undertsand GNU debugger `gdb` commands. The recently enhanced `gdb` can also be used for parallel applications. So, users have at their own disposal two performant tools for debugging not only serial codes but also parallel ones.

Finally, the last release of the Intel C/C++ compilers for Linux offers a set of tools for automatic parallelization of code to be run on multi-core processors or shared-memory parallel computers like Symmetri MultiProcessors. It is called the Multi-Threaded Appliction Support, as cited above, and consists in an infrastructure for the support of OpenMP directives and a tool for the automatic parallelization of segments of code without OpenMP directives for multi-threading programming.

OpenMP and auto-parallelization help convert serial applications into parallel applications, allowing you to take full advantage of multi-core technology like the Intel® Core™ Duo processor and dual-core Itanium® 2 processor, as well as symmetric multi-processing systems:

- **OpenMP** is the industry standard for portable multi-threaded application development because it is effective both at fine-grain (loop-level) and large-grain (function-level) threading (see Section 5.1.2.1); OpenMP directives are an easy and powerful way to convert serial applications into parallel applications, enabling potentially big performance gains from parallel execution on multi-core and symmetric multiprocessor systems;
- **Cluster OpenMP** is an additional tool available with the Intel C/C++ compilers for Linux that allows OpenMP programs to run on clusters; it is a software system that enables a slightly modified OpenMP program to run on clusters based on Itanium® processors or processors with Intel® EM64T, i.e. the Intel 64-bit CPUs not belonging to the standard x86_64 64-bit architecture, the most famous one, developed first by AMD;
- **Auto Parallelization** improves application performance on multiprocessor systems by means of automatic threading of loops; it can be turned on with the `-parallel` option, which instructs the compiler to detect loops capable of being executed safely in parallel and automatically generates the respective multi-threaded code. Automatic parallelization relieves the user from having to deal with the low-level details of iteration partitioning, data sharing, thread scheduling, and synchronizations. It also provides the performance benefits available from multiprocessor systems and systems that support Hyper-Threading Technology (HT Technology), which is a specific technology developed by Intel for multi-threading programming.

Table 4 reports some of the more important optimization options of the Intel C/C++ compilers for Linux (respectively the programs `icc` and `icpc`). Along with the compilers and other tools, partially cited above (idb, OpenMP support, Cluster OpenMP, this last one optional), it is worth noting that the optimization power of such Intel's compilers reside also in the respective libraries, many of which are available as static variants in order to make easier distribute programs compiled with Intel's compilers and to avoid users having to install such libraries too on their computers. For example, the standard Intel math library is contained in the file `libimf.a`: it is an optimized version of the GCC `libm.a`. Codes need to include the header file `mathimf.h` in order to use it. Table 5 shows some of the Intel C/C++ compilers libraries, most of all are present only in the staticcally-linked version, as said previously.

Options to the compilers	Description
`-v`	Verbose compiling, display all steps
`-i_dynamic`	Use dynamically-linked Intel libraries
`-pg`	Create output for GNU `gprof` profiling program
`-tpp6 / -tpp7`	Tune the code for the Intel Pentium II/III and Pentium IV CPU implementations, respectively (similar to the `-mtune=cpu-type` GCC option)
`-axK / -axW`	Enable automatic vecorization (exploitation of SIMD instruction sets) for Pentium III (only SSE) and Pentium IV (SSE2/SSE3) respectively
`-vec_report{0,1,2,3}`	Provide details on code modified by automatic vectorization with four different levels of precision
`-opt_report`	Report on stderr containing details about optimization applied during compilation and/or linking
`-ip,-ipo`	Enable IPO for single files or across file boundaries
`-prof_gen,-prof_use,-prof_file`	Options for switching on PGO

Table 4: a list of optimization options for the Intel C/C++ compilers for Linux platforms, respectively the programs `icc` and `icpc`. Courtesy from [Siemen 2003]

Libraries names	Description
`libcxa.so, libcxa.a`	Intel's own standard library for various C/C++ techniques, e.g. RTTI (Run Time Type Identification)
`libimf.a`	Intel's optimized standard math library (analogous to GCC `libm.a` but with better performance)
`libsvml.a`	Short-Vector Math Library, used for vectorization to run code according to SSEx extensions to the instruction set (SIMD model)
`libirc.a`	Generic library for exploitation of different optimization techniques, among which PGO.
`libcprts.a, libcprts.so`	Intel's standard C++ runtime library
`libguide.a, libguide.so`	Library for parallel programming with OpenMP
`libunwind.a, libunwind.so`	The Unwinder library, which analyzes the stack to trace function calls

Table 5: an incomplete and partial list of Intel C/C++ compilers libraries and their description. Courtesy from [Siemen2003a]

5. 2nd level of HPC: parallel programming for parallel computers

As cited in the previous Sections, particularly in Section 2 and 3, multiprocessor computers are needed when the dimensions of data structures or the time required for the execution of a program are too large compared with the resources of a single PC (being it a modern multi-core system or not !).

As seen in Section 3.3 and respective subsections, multiprocessor supercomputers are classified according two different schemes: the Flynn tassonomy and the way memory resources are used within the supercomputers. Considering that the 4th Flynn's category of parallel computers, the ones implementing a MIMD model, is *de facto* the only one the most manufactured and used supercomputers belong to (except for some special kinds of vectorial supercomputers which are still in used and produced for specific applications), the second criterium for classifying parallel supercomputers is the actual one used for distinguishing the different types of parallel programming paradigms.

It should be remarked and reminded, at this point of this report, the main differences and their respective advantages/disadvantages of the two main types of parallel supercomputers architectures.

Regarding **shared-memory systems**:

- they are based on a bunch of processing units (CPUs with their own cache memories) that shares a common memory space (physical, as for SMPs, or logical, as with NUMA-like clusters of SMPs);
- one change of content in a memory location is visible to all the processing units;
- the exsistence of a global space of memory addresses leads to the possibility of an easy way of programming codes for such systems;
- the condivision of data among different processes (tasks), i.e. instructions flows, is made easier and more fast due to the physical contiguity between CPUs and memory;
- the number of processing units of such systems can not increase beyond certain limits, because with its increase the communication overhead between the CPUs and the memory increases too with a geometric rate (i.e. it converges to a saturation level);
- in order to guarantee the cache coherency for all CPUs, the management of the access to the same memory locations by the different CPUs becomes even more complicated with the increase of the number of CPUs;
- the programmer must control the synchronization of CPUs' accesses to memory locations, in order to avoid that two different CPUs access the same location at the same time.

Regarding **distributed-memory systems** or **distributed shared-memory systems**:

- each computational node has at its disposal its own memory space;
- among different nodes, data can be communicated and interchanged only by communication operations through the network interlinking them;
- there is not a global space of memory addresses common to all the nodes;
- the problem of maintaining the cache coherency does not exist for this system (at least between different nodes) because each node works independently form the others on its own memory devices; cache coherency issues arise inside a node if it is a shared-memory system, like a UMA SMP;
- on each node one or more tasks (or processes) run; if they need some data of another task on another node, the programmer must specifically instructs the code of the parallel program for activating a communication between the two tasks that need to access to each other data;
- the programmer must synchronize the communications between the different tasks;
- the programmer is thus responsible for all the details and issues regarding the communication of data between the tasks;
- sometimes it may be difficult to adapt the layouts of the data structures the program must work on to a distributed-memory model;
- one advantage of such systems is the scalability of memory versus the number of processing units (nodes or single CPUs), the amount of memory increases with the number of nodes;
- each computational node has direct access to its work (RAM) memory, without any overhead due to interferences or cache coherency constraints;
- such systems are cheaper because can be realized by assembling computational nodes (small low cost SMPs) made with commodity components.

These features and differences are at the basis of the different paradigms of parallel programming developed for the two main categories of parallel supercomputers. It should be said that the distinction between the different paradigms and respective software implementations presented in Section 5.1 is not so strict, first of all because the most used and manufactured supercomputers are nowadays highly hybrid systems, as the distributed shared-memory machines. That means that a parallel programming approach specific for shared-memory systems may be, in theory, implemented also for distributed-memory systems. This is the case, for example, of NUMA clusters of SMPs, which are considered shared-memory systems but actually are made of a federation of SMPs (like distributed-memory systems) which can share a global space memory through specifically designed middlewares. The distinction below is perhaps useful from a pedagogical point of view for illustrating the different approaches to parallel programs developed for the different types of parallel machines.

5.1 Parallel programming paradigms

Roughly speaking, there are four main paradigms of parallel programming. The first two have been developed for shared-memory systems; the other ones for distributed-memory machines.

The **shared-memory programming model** has been designed for pure shared-memory machines, like SMPs or for distributed-memory machines with virtual global memory spaces. This model considers for each task the possibility of sharing a common work memory space, other than its own local memory, where they can write to or read from data asynchronously. Locks and “simulated traffic lights” are used for controlling the access to the shared-memory space among the different processes. This programming paradigm does not consider explicit communication directives for managing the transfer of data between the tasks, they are not needed. As a consequence, there is no so much difference between a serial program and a parallel program written according to this model: the development of such a parallel code may be just an incremental process starting from the serial code. Despite this similarity to sequential programming models, this paradigm can not be useful for writing parallel codes which require the subdivision of the global task into many tasks with high degrees of intercorrelations (e.g. intensive data exhange between the tasks or high levels of interdependence between the data manipulated by the distinct tasks).

The other programming model based on a shared-memory hardware architecture is based on the concept of **thread**, which is considered as an execution unit. A program may create different threads that can be scheduled by the OS to be executed by different processing units on a parallel machine or concurrently on a single CPU (multi-threading programming in both cases). Each thread shares with the other ones all the resources associated to the single father process, including its memory space, but may have a private memory space and can communicate with the other ones through the common memory space of the father process. **Multi-thread programming** has been designed for SMPs (virtual or physical) and for multi-core CPUs.

The other two parallel programming paradigms are designed for distributed-memory machines and differ from each other for the necessity or not of sharing data among the different tasks into which the parallel program is subdivided.

The first paradigm is based on the model of message passing: a global task (the parallel code) is subdivided into different sub-tasks assigned to the different computational nodes. If a node contains a single CPU with its own memory, it hosts only one task; if it a SMP, like for distributed shared-memory systems, it can host different tasks, one or more per CPU (according to the fact that it is single or multi-core), all of them running as independent tasks with their own private memory and no common memory, althoug they physically share the same RAM memory.

If the program requires that one task need to know the data managed by another one, a cooperative communication between the two must be activated. This communication let couples of tasks to exchange data between them; it is a cooperative process because one task must “require” some data to the other one, the other must answers and send this data, the first task then must receive it and confirm reception.

This model of parallel program requires that the programmer specifies in the parallel code every details about the communications between couples of tasks, including synchronization of communications in order to let a task having the required data at right time for its processing and doing so for all the tasks (a problem that is included in the so called load balancing issue among many other ones). The programmer must control explicity, at low level, the inter-tasks communications. The technological solution for doing this is a library of functions/subroutines to be used within the code. **MPI** (**Message Passing Interface**) is a standard for implementing such a library of functions/subroutines and the respective API. Different implementations of MPI have been realized by public Research initiative or private vendors/manufacturers of supercomputers. Section 5.1.1 presents a brief overview of the MPI API.

The second distributed-memory parallel programming model is called **data parallelism** and is not at all a specific technique for parallel programming on a specific architecture. It is used when the same bunch of instructions of a code must be executed on a set of data, for example on every element of a data structure, for example multi-dimensioanl arrays. In these cases, a multi-thread programming model or the MPI one can be used to realize a parallel program which is a set of replicated tasks, each of each working on different chuncks of the data structure. The example of Figure 26 may be considered as a case of data parallelism implemented by OpenMP programming.

Some programming languages as Fortran 95 and High Performance Fortran (HP Fortran) offer to the user specific construct to manage data parallelism within codes.

Both shared-memory and distributed-memory parallel programming models (and their respective implementations) refer to a specific type of runtime model called **Single Program Multiple Data** (**SMPD**): in both cases, there is a parallel program which generates different tasks; all the task may execute the same instructions of the program or only some of them (different from task to task) but it is the same program replicated for each task. Within the program there may be instructions that lead the task to execute all the program or only some parts of it. Each task can work on all the data of the program or only on a bunch of them. At a certain runtime step, the task may be executing the same instructions or different ones.

The other runtime model of execution is called **Multiple Program Multiple Data** (**MPMD**) and consists in the execution of different programs as different tasks on different computational nodes. A program may be executed by more than one task and different tasks can work on different data as well as same data. This runtime paradigm is used in distributed computing, for example in grid computing.

5.1.1 Message passing paradigm and the Message Passing Interface (MPI)

As previously cited, MPI is a standard for writing an API and a library of functions/subroutines that implement the message passing parallel programming model. It was realized by the interaction of different partners from the industry and scientific Research worlds. The first proposal for the realization of such a standard came in April 1992 at the Workshop on *Standard Message Passing on Distributed Memory Environment*, held at the Center for Research on Parallel Computing, Virginia. The first draft of the standard was elaborated by a group of the Oak Ridge National Laboratory, USA, on November 1992. Then, different partners (Research institutions, hardware and software vendors) decided to create a common forum, called MPI Forum, for sharing the suggestions and needs of different user communities in developing the standard. Finally, on May 1994, the first draft of MPI-1 was released. Then new updates and releases followed up to the MPI-2 standard (1996). The documents of both versions of the standard are hosted on the official MPI Web site at the Argonne National Laboratory, USA. The work on the MPI standard still goes on in order to address issues of distributed-memory systems programming that were not considered in the previous years because they have been arising in the meantime with new software and hardware developments in the field of parallel computing.

The MPI-1 standard defines the prototypes of functions/subroutines for realizing message passing programming within C/Fortran 77 codes. The details on how these functions/subroutines are implemented and collected in a library are left to the different Research institutions or vendors which want to realize and/commercialize an implementation. The standard defines the name, the arguments, the returned data and the sequence of calls of each function/subroutine of such a library. Each implementation must at least follow such standard; then, it may offer to the user other functions/subroutines for supporting other message passing programming functionalitities.

The MPI standard version 1 does not consider how a MPI-instrumented parallel program must be lunched (it is an issue connected with the implementation and its installation on a specific machine), how a MPI-nstrumented parallel code can be debugged, how to change the number of tasks at runtime. The MPI-1 standard also does not include parallel I/O functions/subroutines. The MPI-2 standard addresses the last two issues along with support for C++ and Fortran 90 programming.

Nowadays, all MPI implementations are MPI-1 compliant, while not all of them are MPI-2 compliant, only a small fraction of them have been improving their releases in order to include functions/subroutines for the parallel I/O and the dynamic management of tasks.

A MPI parallel program runs according to the SPMD runtime model: the code of the program is replicated *N* times and assigned to *N* tasks, each of them has a number of identification called **rank**. Different tasks can execute different instructions of the code because within the code `if` constructs can be inserted in order to make blocks of code be executed by one specific task (one instance of the running program) only if the rank of that task has a certain value. The rank of the task can be assigned to a variable of the program, called for example `MyRank`, calling within the program a special function/subroutine of the library.

The MPI standard extablishes also the syntax of the functions/subroutines: for C functions, their prototypes and defintions **must be** of the type `err = MPI_`<name of the function> `(`<parameter1>,<parameter2>,.....`)` where `err` must be an `int` variable used as an error-of-execution code, <parameter1>, ..., are input or output parameters (in the second case they are passed by reference, i.e. using pointers to them) of the function and its identification name must always begin with a capital letter and be preceeded by the string `MPI_`. Figure 24 shows a typical structure for a MPI-instrumented C parallel code, with the calls to the basic MPI functions for initializing and closing the parallel environment, defining the basic environment of communication between the tasks, obtaining the rank of the individual task, making that task execute a specif bunch of instructions reserved to it.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <mpi.h>
4 int main(int argc,char* argv[]){
5    int MyRank,size;
6    /*Initialize MPI environment*/
7    MPI_Init(&argc,&argv);
8    /*Get the rank of the task*/
9    MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);
10   /*Get the total number of task in the MPI_COMM_WORLD communicator*/
11   MPI_Comm_size(MPI_COMM_WORLD,&size);
12   /*Execute the instructions, including communications of data*/
13   .........
14   if (MyRank == x){
15   ....
16   }
17   .......
18   .......
19   printf("Task %d of %d \n",MyRank,size);
20   /*Finalize the MPI parallel environment*/
21   MPI_Finalize();
22 }
```

Figure 24: example of the structure of a generic MPI parallel C code. The first parallel instruction is the pre-processor `#include` instruction for including the MPI library header file, mpi.h (line 3). Line 7 and 21 contain respectively the calls to the MPI function necessary to initialize and terminate a parallel environment. They can be called only one time within a code and every call to other MPI functions must be contained in the lines delimited by these two functions. In blue (lines 9 and 11) are showed two important MPI functions used to obtain information about the communicator used in the parallel program , in this case the default one initialize automatically with the `MPI_Init(...);` function call. The block of instruction in red character shows a part of code that is executed only by one task, the one whose rank satisfies the conditional expression, while the lines of code in violet are executed by each task. It should be remembered that the overall code is cloned N times and executed by the N tasks (processors) synchronously.

First of all, it can be seen in Figure 24 that a specific header file, called **`mpi.h`**, is included in the program to be pre-processed. It is the header file of the MPI implementation library; it contains the prototypes of all MPI functions plus the definition of some specific MPI built-in data types. Some of these data types corresponds to C basic data types but have only a different name in order to be used within the arguments lists of MPI functions. For example, a variable defined of the type `MPI_Int` corresponds to a C `signed int` variable, one defined as `MPI_Float` is of type `float`, etc. MPI built-in data types are defined and implemented in order to hide the details of, e.g., the floating-point representation, which is an issue only for who realizes the MPI implementation. MPI allows for automatic translation between representations in a heterogeneous environment. As a general rule, during a reciprocal communication operation between two tasks, the MPI data type of a variable used as argument of a send instruction must coincide with the MPI data type of the corresponding variable used in the respective receive instruction of the receiving task (see below for communications between task and how they are implemented through functions calls). The identification names of MPI built-in data type follows the same syntax rule for the names of fucntions: they must begin with the string `MPI_` and then it must follow the name of the data type with the first letter being capital. These are details, however, that still interest only the implementer of the MPI library. The programmer must only take care of using the correct syntax when calling MPI functions and using MPI data types.

mpi.h also includes the definition of certain constants, macro and MPI built-in data structures used for communications operations and referenced as **MPI handles**. Handles are returned as output by various MPI calls and may be used as arguments in other MPI calls. In C parallel programs, handles are pointers to specially defined data types (created via the C typedef mechanism). An example of handler to a MPI built-in data structure is the MPI_COMM_WORLD communicator. A communicator is a MPI built-in data structure used to define an "environment of communication" for the tasks. In C, MPI communicator are defined as variables of the MPI built-in data type MPI_Comm. Many different communicators can be defined by the calls of specific MPI functions. Each communicator can include all the tasks or only a subset of them. Each communication function called within the code must, at least, have as one of its arguments the handle of one communicator. Two different tasks may execute a reciprocal communication (send/receive) if and only if they belong to the same communicator. MPI_COMM_WORLD is the handle for the standard communicator including all the tasks of the parallel program and it is defind automatically when the MPI parallel environment is initialized.

There are two MPI functions which initializes and concludes the parallel environment, respectively MPI_Init(...) and MPI_Finalize(). Both of them must always be inserted in a MPI-instructed C parallel code and must compare only one time. MPI_Init(...) must be the first MPI function to be invoked in a parallel program. It initializes the parallel environment and sets up the default communicator, whose handle is MPI_COMM_WORLD. MPI_Finalize() extablishes the end of the parallel environment, no other MPI function/subroutine can be called after it. It concludes all the communications between the tasks and free the memory spaces used as buffers for the communications.

In Figure 24, it can be seen that at line 7 the parallel environment is initialized by invoking the function **MPI_Init(&argc,&argv);** , which may have as arguments the ones passed to the main function. Initialization of the parallel environment means that from that line on the code is replicated *N* times and assigned to the different functions. **Only after** the call to that functions, other MPI functions can be invoked. Two important examples of MPI functions are called at lines 9 and 11 and both of them are used to obtain information about the used communicator, which in that case is the default one with the cited handle MPI_COMM_WORLD. At line 9, the call **MPI_Comm_rank(MPI_COMM_WORLD,&MyRank);** assigns to the variables MyRank the value of the rank of the task that is executing its copy of the code and this happens for all tasks which the parallel program (the father task) has been divided into. At line 11, the call **MPI_Comm_size(MPI_COMM_WORLD,&size);** determines the total number of tasks within the indicated communicator, which in the case of use of MPI_COMM_WORLD means the total number of tasks, and assigns this value to the variable size. As can be seen and as said previously, the output of MPI C functions are defined as arguments of the functions themselves and are passed to them by reference, that is their pointers are passed by copy of values in order to be able to change their values. The output of MPI functions must be defined before their calls as varibales, as happens at line 5 in Figure 24 in the case of these two functions. Both size and MyRank are variables of type signed int in C programs. MyRank begins from 0.

Each task executes the same instructions of the other ones, as said previously, unless some instructions are within the body of a conditional construct (e.g. a if construct) that selects that bunch of code lines only if MyRank assumes a certain value, i.e. only if the task has a certain identity. In figure 24, all the lines with instructions in magenta colour are executed by each task. For example, if the total number of task is N, the output instruction at line 19 will be executed by all the task and on the stdout of the shell the sentence **Task n of N** will be printed N times, each time with a different value for n. However, the lines of code from 15 to 16 will be executed by only the task with rank equal to the value of the variable x. This is the general method through which MPI implements the work-sharing among different tasks. It is conceptually very simple but let the programmer all the responsibility to take care of it. For example, if the programmer wants its MPI parallel code execute the same instructions but on different parts of a data structure or different data structures, as multi-dimensional arrays, he/she have to explicitly use an if construct as at line 15 of Figure 24, insert within it the same instructions but repeat it many times with different control conditions on the MyRank variable value in order to make the same instructions work on different bunch of data. How to select the different bunch of data is an issue to be assessed by the programmer him/her-self within the bodies of the different if construct. A C switch construct may be used instead of a repetition of different if ones.

If the different tasks do not need to use values of data managed by other tasks, the parallel code implements a form of data parallelism, otherwise communications between the tasks are needed, because each variable in the code is replicated as many times as the numbers of tasks and it has a different memory location for each task, into different memory devices (the local memories of each computational node on which the task runs).

The MPI library offers different types of functions/subroutines for communications of data between tasks. The implementation of the communication is one of the most difficult point in writing a correct MPI parallel program first of all because the programmer have the complete control of the communications at low-levels.

There are two main types of MPI inter-task communication models: the first one is called **point-to-point communication** and involves only a couple of tasks; the second one is called **collective** and may involve groups of tasks or all the tasks within a communicator. As previously reported, two or more tasks can communicate (transfer) data across their memories only if they belong to the same communicators.

Point-to-point communication is a direct communication between two tasks, one of which sends a “message” while the other one receive it. It is a two-sided communication: an explicit send instruction inserted in the piece of code run by the sending task must have a correspondent receive instruction in the piece of code run by the respective receiving task, otherwise data can not be transferred.

In a generic send/receive communication, a message consists of an “envelope” of data, indicating the source task identification parameter (for example the rank within the communicator) and the destination task one. Then, the message contain a body, that is the set of actual data to be transferred from one task to the other (physically from one memory location on a computational node to another memory location on another computational node).

MPI uses three pieces of information to characterize the message body:

- the buffer, i.e. the address of the starting point in the memory of the sending task where outgoing data is to be found for a send instruction or where incoming data is to be stored for a receive one; in C, buffers are usually pointers;
- the data type, i.e. the type of data to be transferred, expressed using MPI built-in data types names;
- the count, i.e. the total number of items of the indicatd data type to be transferred.

It has to be noticed that the use of MPI built-in data type names as arguments of functions/subroutines, implementing send/receive instructions, guarantees that the programmer don't have to explicitly worry about differences in how machines represent data, e.g. differences in the representation of foating point numbers. This issue is not relevant when the distributed-memory parallel computer has homogeneous nodes, as it happens in the case of clusters of SMPs. However, it assumes a certain importance in the case of an heterogeneous cluster, with nodes being, for example, different desktop PCs.

MPI provides a great deal of flexibility in specifying how messages are to be sent, implementing different communications modes that define the procedures used to transmit messages as well as a set of criteria for determining when the communication event (i.e. a particular send/receive operation) is complete.

There are four distinct types of send communication modes:

- **standard send**;
- **synchronous send**, which is defined to be complete only when receipt of the message at its destination has been acknowledged;
- **buffered send**, which is considered complete onl when the outgoing data has been copied to a local memory buffer, without any implication about the arrival of the message at the destination task;
- **ready send**.

A send instruction of each type is implemented via a specific MPI function/subroutine. In all cases, the completion of the communication operation implies that it is safe to overwrite the memory locations where the outgoing data were originally stored.

MPI provides only one receive communication mode. A receive operation is considered complete when the incoming data has actually arrived a the destination task and it is available for use.

In addition to the communication mode, MPI categorizes send/receive instructions (and consequently the functions/subroutines implementing them) also as **blocking** and **non-blocking** communication instructions. A blocking send or receive function/subroutine does not return from the function call until the communication operation has actually completed. Thus, it is ensured that the relevant completion criteria have been satisfied before the calling flow is allowed to proceed. With a blocking send, for example, the programmer can be sure that the variables sent can safely be overwritten on the memory area of the sending task. With a blocking receive, the programmer is sure that the data has actually been stored in the indicated memory locations of the receiving task and is ready for use. A non-blocking send or receive returns immediately to the calling flow, with no information about whether the completion criteria have been satisfied. Although they are dangerous, non-blockign send/receive instructions imply the advantage that the processor is free to execute other instructions while the communication process proceeds in the background.

A send instruction (i.e. its implementations in a function/subroutine) can then be of eight different types if all the combinations between the two modes are considered. For example, a nonblocking synchronous send instructions returns immediately although the send will not be complete until receipt of the message has been acknowledged. The sending task can then proceed with other instructions, testing later if the send is complete or not. Until it is complete, the programmer can not assume that the message has been received or that the variables to be sent may be safely overwritten, so he/she should avoid to update their memory locations.

In addition to point-to-point communications between individual pairs of processors, MPI includes functions/subroutines for performing **collective communications**. These functions/subroutines allow larger groups of tasks to transfer data betwen them according to different paterns of communications. All the tasks participating in a collective communication may belong to the same communicator too and the communications instructions remain reciprocal in the sense that at one send instructions from a task it must correpspond a receive one into another one.

The main advantages of using collective communications functions/subroutines over building the correspondent point-to-point ones are:

- the reduction of errors, because a communication operation can be implemented within the code with only one function/subroutine call insted of many lines necessary for repeted poin-to-point ones;
- the source code is, thus, much more readable, then its debugging is easier;
- optimized forms of the collective communications functions/subroutines often run faster than the equivalent operations expressed through successions of point-to-point ones.

Examples of types of collective communications operations are the **broadcast operations** (one task sends some data to all the other ones of the same communicator and the receiving tasks positions these data, within their memory, in the locations corresponding to the ones occupied in the memory of the sending task), **gather operations** (data collected in the memory of a single task are subdivided in many chunck which are then sent to all the other tasks belonging to the same communicator), **scatter operations** (the same as gather ones but in the reverse order), **reduction operations** (a single task, called the root task, collects data sent by other tasks in a group and then comnines them into a single data item applying on them different possible type of operations, like, e.g. sums, maximum, minimum, etc. ...).

For reasons of space and time, the bindings of the functions/subroutines implementing all these communications instructions are not reported here. They can be found on MPI-1 and MPI-2 reference guides, books or directly at the URL of the two versions of the standard, at the official MPI Web site.

Also for reasons of space and time, in this uncomplete report other important issues of MPI parallel programming like communicator/communications topologies, how to avoid deadlocks in the communications between tasks and parallel I/O are not treated but the bibliography (Section 7) and the list of useful Websites (Section 8) offer good references for the interested reader.

Coming back to the example of MPI parallel code showed in Figure 24, the parallel environment is closed by the call to the `MPI_Finalize();` function/subroutine, after that no other MPI functions/subroutines can be invoked.

5.1.1.1 MPI implementations

There exist many MPI implementations. As cited in Section 5.1.1, all of them nowadays are fully compliant to the MPI-1 standard, while only few of them adhere completely or partially to the MPI-2 one. Many implementations are proprietary libraries, while we make here only references to Open Source/free implementations realized by academic institutions or mixed groups made by academic institutions and industrial hardware and software vendors. An extended list of both types of implementations are maintained by the Mathematics and Computer Science Division of the Argonne National Laboratory, USA, which maintains also the official Website of MPI.

MPICH is one of the most known implementations. It has been developed and updated during the years by a group of researchers at the Mathematics and Computer Science Division of the Argonne National Laboratory, led by W. Gropp and E. Lusk, in collaboration with A. Skjellum of the Department of Computer Science and Applications, NSF Engineering Research Center for CFS, Mississipi State University. The acronym MPICH stands for “MPI Chameleon”: the founders of the implementation chose the chameleon as the main symbol for their implementation because since the beginning they had in mind to realize an implementation of MPI that would be much portable as possible and the chameleon symbolizes the best animal that is able to adapt to one's environment.

MPICH has grew fast during the years, since 1994, according to the phylosophy of Gropp and Lusk of offering to the users an implementation that tracks the evolution of the MPI standard. Many versions have been released since 1994-1995, every time the MPI Forum has released a new version of MPI, a new version of MPICH has been released too. Gropp and Lusk has managed to obtain such a result. MPICH was released with the version 2 in 2005. During the years it has been widely used within the scientific Research communities. The philosophy of Gropp and Lusk at the basis of their implementation is not only to offer to the users constantly updated and improved versions of the library and parallel tools but also to submit to the MPI Forum, which control the evolution of MPI standard, issues and problems met by the implementors. This feedback circuit between MPI standard evolution and MPICH implementation update has been very successfull. Many other projects and MPI implementations has been developed based upon MPICH. Other projects did not succeed in the same way because they waited too long for a stable version of MPI standard, while the MPI standard itself is in continuous evolution.

Another successful implementation is called LAM/MPI, developed at Indiana University and now converged in a recent big project for the realization of an implementation that is fully compliant to the MPI-2 standard, OpenMPI. OpenMPI derives from an initiative based on the collaboration between Indiana University (LAM/MPI), University of Tennessee (FT-MPI), Los Alamos National Labortory (LA-MPI), Sandia National Labs (USA) and the HPC Center at Stuttgart (PACX-MPI). This implementation actually combines four other ones developed by the different partners participating in the project: LAM/MPI, FT-MPI, LA-MPI and PACX-MPI.

While MPICH still remains perhaps the most portable implementation of MPI, the project OpenMPI seems to be very promising for the realization of a full integrated MPI-2 parallel programming library and toolbox. Whichever implementation among the ones cited above the programmer/scientist chooses to use, MPI has been and will be for long time one of the most used tools for parallel programming, specifically for distributed or distributed shared-memory supercomputers. It is worth remarking the features that have made MPI and its implementations more successful than other tools like HPF (which is dedicated to data parallelism bu not only).

However, it should be useful to remember when MPI parallel programming may be used:

- when it is necessary to develop a performant parallel code for a distributed-memory machines or an hybrid one, with a high degree of portability;
- when it is necessary to develop libraries for parallel computing;
- when an application implies dynamic and irregular relations between different sets of inter-dependent data so that it can not be treated with a data parallelism approach;
- when the performance of the parallel code is an important asset for the intended application.

In the last point, the word performance mainly refers to a low level of latency in the communications operations: a parallel program is considered performant if the time spent by the execution of communications operations is not too high in respect of the actual separate elaborations made by the different tasks.

5.1.2 Shared-memory systems paradigms

The shared-memory programming model has been superseded by multi-thread programming, which is the standard model for programming shared-memory machines. The multi-threaded model implies the use of synchronization operations necessary to avoid that many threads update a shared-memory space location at the same time. It has been implemented with libraries of primitives and sets of compiler-directed instructions to be inserted into the sequential code of a program.

Historically, two main projects have tried to realize a standard for multi-thread programming. The first one is the **POSIX Threads** model, usually known as **Pthreads**. It was standardized as IEEE POSIX 1003.c in 1995. It includes a library of C codes, which has served as the reference frame for the realization of other proprietary libraries by different vendors. The second project is called **OpenMP** and is the result of a standard defined and realized by a group of software and hardware vendors. The OpenMP API (Application Programming Interface) for the Fortran language was released at the end of 1997, while the one for C/C++ at the end of 1998. This multi-thread programming model API is based on compilation directives to be inserted in the source code of a sequential program. This feature lets the user write parallel programs for shared-memory machines starting directly from a serial code, so the entire process of parallel programming can be divided into incremental steps, while with the message passing model a parallel program must be designed accurately with an algorithm that is sometimes completely different from the one of the original serial code.

In order to understand how multi-thread programming does function, some notions about OSs have to be introduced. The type of OSs considered are the ones belonging to the Unix families.

First of all, the concept of “process” (sometimes called task in this report) must be defined and analyzed properly in order to understand the basis of multi-thread programming. When a program is run, the OS assigns to the program certain resources. The set of such resources, i.e. the context within which the program runs, is called a process. First of all, some memory space is assigned to the program and to the user that has lunched the program. This memory space is subdivided into different regions each of which is dedicated to different data needed by the program. Figure 25 shows a schematic example of the user space memory dedicated to a process.

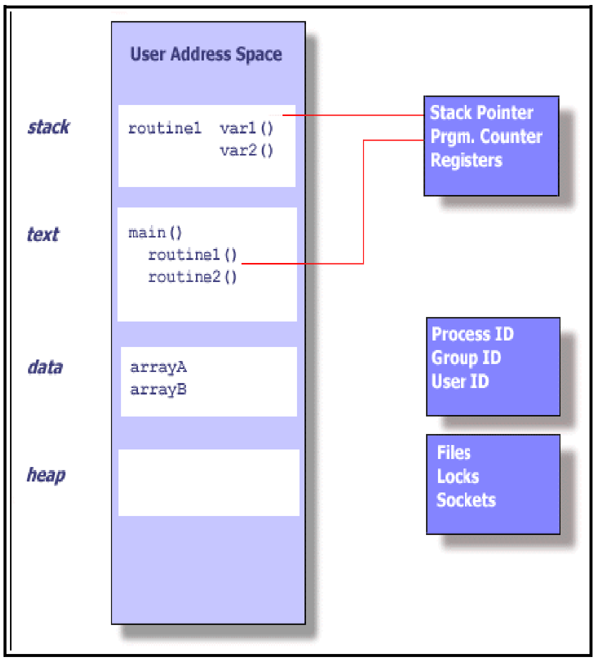


Figure 25: schematic overview of the memory space assigned to a process run by a specific user. This memory space and the CPU registers dedicated to the process constitutes the context within which the program is executed. The user memory space is subdivided into different parts having different roles. The figure shows that the text area contains the instructions of the program (here represented in pseudo-code), while the data area is dedicated to the allocation of the variables and data structures of the main function/subroutine. The local data used by functions/subroutines called from the main one are allocated in the stack area: when a function is executed, such data are removed from the stack according to a FIFO management model (the stack can be visualized as a pile). Finally, dynamically allocated variables or data structures reside in the heap space. If the list of instructions of the program is executed by only one processing unit (CPU) or there is only one flow of execution, then it is said that only one thred (instance) of the program as been generated by the OS. Figure courtesy from S. Meloni, Parallel Programming with OpenMP, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

The **text memory** slot is dedicated to storing the instructions of the object code of the program. A special CPU register, called the **Program Counter**, tracks the point at which the execution of the program is at a certain time. The program counter is an important resource of the running process. If there is only one execution flow the list of instructions of the code goes through, then it is said that the process is run only with one thread (instance). In the example of Figure 25, that means the functions routine1() and routine2() are executed sequentially by only one CPU. Other important resources constituting a process are: the **stack memory** slot, i.e. the part of memory assigned to a process where only local variables of functions/subroutines called within the program are allocated, according to a FIFO (First In First Out) model of memory management; the **stack pointer**, which tracks the last used memory allocation in the stack space; the **heap memory** slot, dedicated to dynamic allocation of data, which depends on input data during runtime; the **data memory** slot, i.e. the segment of memory dedicated to local variables of the main program which are statically allocated (this memory space is subdivided into two different parts called **data** and **bbs**, the first one for initialized data, the second one for not initialized ones).

When two or more execution flows of the same process are assigned by the OS, it happens that different parts of the list of instructions in the text memory segment are assigned to different execution flows; a program counter is assigned to each flow and the stack memory is subdivided into distinct parts which refers to the different parts of the list of instructions. In this case, the process is run in a multi-threaded way (many instances). Figure 26 shows an example of multi-threaded process: different functions called in the main program are assigned and executed to the different threads, the corresponding stack memory segments become private memory space of each thread while the data and heap memory slots remain global memory locations accessible by each thread. In a shared-memory parallel computer, the threads are run by different CPUs, for example. However, multi-threaded processes can be run also on a single CPU with dedicated units in order to manage virtually the different threads as if many processing units do exist (Intel has developed a technology called HyperThreading for some of its CPUs). The different program counters correspond to the threads and they track the execution of those specific parts of the process. At each step of the running process, the program counter of a thread refers to a specific instruction of the code which has access to the memory locations of its private stack segment and to the data, bbs and heap segments, which are accessible also by the instructions of the other threads. According to this model, a multi-threaded process is tailored to run on a shared-memory architecture, being it a single CPU built with special technologies for multi-threading (as in the case of Intel Pentium HyperThreading or Multi-Core technology) or a SMPs (a physical shared-memory system) or a NUMA cluster of SMPs (a logical shared-memory system).

Pthread is the standard tool for multi-thread programming on Unix-like computers, being them parallel or not. As said previously, it is based on a library of C-coded functions for managing all the functionalities needed by multi-threading. OpenMP has managed to become the *de facto* standard for parallel programming on shared-memory machines. It is based on compiler-directed instructions (to be inserted into the code) for the synchronization of the thread flows and the work-sharing among them, on a library of functions/subroutines for obtaining the values of some parameters of the parallel process (e.g. the total number of threads) and on some Unix-like environment variables for the definition of the context of the process itself. Due to its diffuse use within both the industrial and academic communities, the following sub-section is dedicated to an overview of how OpenMP functions and how parallel programs are developed according to its directives.

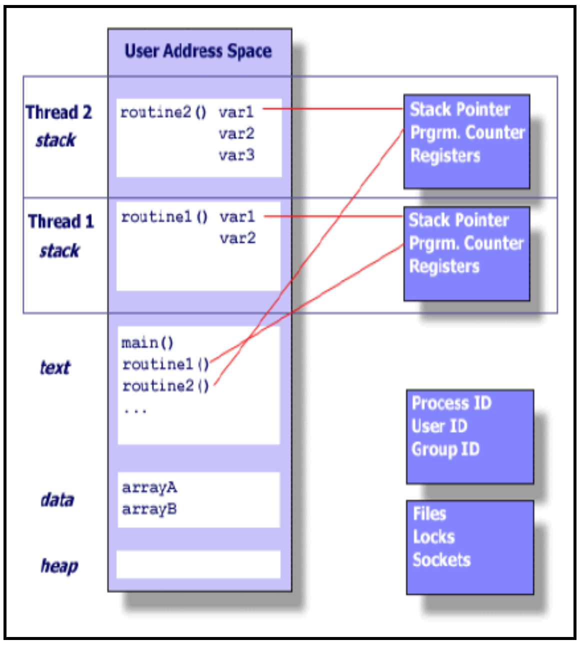


Figure 26: example of multi-threaded process/program. The list of instructions in the text memory area is subdivided into different chunks and associated to different program counters that track their execution. In this example, calls to two different subroutines from the main one are associated to the different threads. Each thread has its own stack memory area, which is unaccessible to the others, while the data memory locations are accessible to every thread. The data area constitutes a global memory addressing space. The distinct threads may be executed by different CPUs, as in the case of SMPs, or by a single CPU which support the multi-thread programming model by a specific hardware design (as the Intel Hyper-Threading technology on many of its recent CPUs). Figure courtesy from S. Meloni, Parallel Programming with OpenMP, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

5.1.2.1 OpenMP

As previously said, an OpenMP parallel program is an instructed serial code containing special directives that can be understood only by the compiler. Each of this directive is signaled to the compiler by a specific keyword, a **sentinel**, that the compiler can recognize only if the compilation is done with the option `-openmp` (for most of the compilers, e.g. GCC's, Intel's ones). It means that the OpenMP library of directives and functions must be installed within the compiler itself. The last releases of the Intel C/C++ compiler for Linux, for example, already includes OpenMP within its package.

The OpenMP directives have two main roles:

- delimiting a block of code that must be treated by the compiler according to the multi-thread programming model, that is it must be treated as a parallel code run by many threads;
- indicating with specific **work-sharing keywords** or **directives** how to assign to the different threads the instructions of that block, in order to be executed concurrently, i.e. how to realize a parallelization of that block of code.

At runtime, when the program starts, it is executed only by one thread, called the **master thread**. When an OpenMP directive is reached, the master thread is splitted into different concurrent threads executing the same list of instructions if any work-sharing keywords have not been specified after the directive or executing different instructions according to the specified work-sharing keyword.

When the block of code delimited by the OpenMP directives has been executed by all the threads, the control comes back to only one thread, the master one, up to meeting another parallelization directive along the flow of the master thread itself.

This runtime model is called **fork-join**. According to this model, the programmer has only to specify the parts of his/her code that must be run in parallel and to choose which type of work-sharing directives must be applied to that part of the code. For different constructs or list of instructions, OpenMP offers different work-sharing directives that automatically instruct the compiler on how to realize the parallelization.

How many threads to create at each fork is a parameter that can be fixed after compilation, before running the program as usually done with serial ones. An environment variable, `OMP_NUM_THREADS <n° threads>`, must be fixed with the Unix-like instructions

```
setenv OMP_NUM_THREADS <n° threads>
export OMP_NUM_THREADS <n° threads>
```

where `<n° threads>` is a number equal to the total number of threads to be created from the master one (which is included among them). Otherwise, some functions of the OpenMP library can be used within the code itself, before the block of parallel code, that is before an OpenMP parallelization directive: for example, the call to the function/subroutine `omp_set_num_threads(N);` in a C code sets the number of thread to `N` while the call to `omp_get_num_threads();` returns the number of threads used for the parallel execution of a block of code it is inserted within.

```
1 #pragma omp paralell
2 {
3  for  (i=0;i<n;i++)
4     a[i]=b[i]+c[i];
5 }
```

Figure 27: an example of a simple C code construct which is parallelized automatically at compiling time by the OpenMP directive `omp parallel` preceeded by the OpenMP sentinel `#pragma`. In this case, a simple for loop is parallelized in the simplest way: the same loop construct is executed simultaneously by many threads, all of them accessing the same variables `a[]`, `b[]`, `c[]`. This an unuseful parallelization which slows down the program because the different threads repeat the same vectorial sum between the same arrays.

Figure 27 is an example of block of C code (a `for` construct) which is preceded (line 1) by an OpenMP directive of parallelization: the sentinel, for a C code, is the string `#pragma` and it is a kind of “flag” recognized by the compiler that instructs it to consider the subsequent code lines, between the two graph parentheses, as the ones of a parallel code; what follows is the standard OpenMP directive for creating a parallel environment for the subsequent block of coce, i.e the directive `omp parallel`. This parallelization directive realizes the fork of threads at runtime, that is it creates N different threads from the master one (including it, so N-1 new threads) that execute the subsequent block of code. The execution of that parallel block of code follows according to the model SPMD: each thread executes the same instructions. In the example of Figure 27 each of them execute the `n` iterations of the `for` loop.

From the point of view of parallelism, the previous case shows that the parallelization does not bring any performance increase. The parallelization of a loop is useful when the instructions of the loop have some degree of independence and the total set of iterations can be subdivided into different chuncks, each of which includes only a subset of iterations. In this case, the model of parallelism is the fourth one of the list presented in Section 5.1, i.e. data parallelism: the same instruction, a sum between elements of two different arrays, can be executed on the different couples of values by distinct processes, threads in the case of a shared-memory machine. In order to implement data parallelism in the case of Figure 27 or in general in order to instruct the parallel program to distribute the work among different thread avoiding they do the same work, OpenMP, as previously said, offers a set of work-share directives to be associated to the one for creating a parallel environment for a block of code.

Figure 28 shows an example of work-share directive dedicated to the automatic parallelization of a loop construct. The whole code calculates a matrix-vector product. The loop on the index `i` can be parallelized (i.e. vectorized), in order to exploit the advantage of memory chaching and the independence of the instructions to be executed within it (i.e. the independence between the elements of the result 1D array `u[]`). For doing so, the directive `for` has been added to the one `omp parallel`. Sometimes, it is said that the OpenMP directive for parallelization of the loop is `parallel for`, as the result of the association of the two directives `parallel` and `for`. Anyway, the result is that when the program, at runtime, arrives at the `#pragma omp parallel for` it created N threads each of which executes the body of the first for loop for a limited number of iterations, i.e. for a sub-interval of values of the index `i` and each thread treats a different sub-interval such that when the loop has been executed and the flow re-join into the master thread, all the iterations on the index `i` have been executed.

All the variables within the parallelized block of code are considered as shared variables, so they are accessible to all threads, including the loop index i, which is changed concurrently by each thread. There could be some slowdown of the run time due to the fact that each thread has to change the value of the index `i` in order to access different elements of the arrays. In that case, the variable `i` can be defined as `private(i)` within the parallel environment: each thread will have an independent copy of it allocated in its private stack memory area and that copy will assume the right values assigned by the work-sharing directive. When the parallelized block of code terminates, the private variable gets shared again with the same value she had before the fork process.

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
int main(int argc, int** argv)
{
   double A[1000][1000],v[1000],u[1000];
   int i,j;
   for (i=0;i<1000;i++)
   {
      v[i]=(i+1)+0.001;
      u[i]=0.0;
      for (j=0;j<1000;j++)
      {
         A[i][j]= double(i)+j+0.001;
      }
   }
 #pragma omp parallel for
 for (i=0;i<1000;i++)
 {
   for (j=0;j<1000;j++)
   {
      u[i] = A[i][j]*v[j];
   }
 }
```

Figure 28: an example of a C program instrumented with OpenMP dirctives for the parallelization of a for loop construct in a useful manner, according to the parallel processing model of SPMD (Single Program Multiple Data), which realizes the data parallelism: the same instructions are executed by different threads on distinct independent parts of data structures. The OpenMP work-sharing directive that does so is `parallel for`, although only the keyword `for` has been added in this code in respect of the ones of Figure 26. However, the combination of the two directive `parallel` and `for` creates a new one, `parallel for`, which implements the work-sharing strategy for the parallelization of the loop. It should be noted that the parallelization involves only the first loop, the one with the index `i`. Nested parallel-for directives are allowed so in this case also the internal loop on the index `j` could be parallelized according to the data parallelism model. However, in this case, a nested parallel for is not efficient or does not improve anymore the performance of the code because each for loop with index i fixed is run fast by each thread due to the exploitation of spatial and temporal locality, so of the features of cache memories. The example is courtesy from S. Meloni, Parallel Programming with OpenMP, Lecture Notes for the 2nd edition of the CASPUR Summer School on Advanced High Performance Computing, Castel Gandolfo (Roma), august 28th – september 8th 2006.

As previously cited, OpenMP offers many work-sharing directives for the automatic parallel management of special constructs as loops, conditional constructs, etc. ... Other important directives are implemented for solving problems of concurrency among the different threads and for obtaining synchronization of their read/write operations on the global accessible memory area. Other directives are dedicated to the management of the data allocated in the stack area of each thread, the private data. The examples of Figures 27 and 28 are very reductive compared to the great variety of multi-thread programming tools that OpenMP offers to the programmers, but these two examples already catch and show some important features of OpenMP parallel programming of shared-memory systems: as it can be seen, the parallel environment directives let the programmer to develop its parallel program starting from the serial version of the code and then to incrementally tackle those parts of the code that are more critical (from the performance point of view) and that can take advantage from a parallelization and run on a shared-memory machine. The work-sharing directives are the other great asset of OpenMP parallel programming: they are very compact and let the user avoid to consider many technical details about the parallel management of a block of code; these details are treated automatically by the compiler supporting the OpenMP API. In comparison to MPI programming, this means more abstraction and high level directives for parallel programming but even less control on the behaviour of the parallel code itself.

The OpenMP initiative has, in fact, among its objectives, the one of becoming a standardized (maybe by IEEE ?) easy-to-use set of tools for shared-memory systems parallel programming, implementable on different architectures and platforms (high portability). This aim implies the use of high-level an user-friendly tools and methods at the cost of loosing in possibility of customization by the user to its specific parallelization problem.

The nowadays outset of the Multi-Core technology by Intel, AMD and others CPU manufacturers and vendors may be pushed forward the development and standardization of OpenMP API, because multi-core CPUs with an increasing number of cores will need multi-thread programming environments in order to let the users exploit their computationl potentialities.

6. Concluding remarks and aknowledgements

High Performance Computing has been playing an increasing role in natural Sciences. Although parallel supercomputers remain the essential tools for solving complicated computational problems, scientists can get benefits from the continuous advances in hardware and software technologies. This reports would like to be a short (thus incomplete) review of techniques for developing high performance scientific computing codes, both in the serial and parallel cases. Serial codes may take advantage of the increasing power of modern desktop PCs if the codes are written according to techniques able to exploit their hardware features. Parallel codes are strictl necessary when the resources required by a computational problems are very large. In both cases, optimization techniques and tuning are essential tools for reducing the runtime of a code and to obtain a better use of CPU/memory resources.

Optimization necessarily can be implemented only knowing a little bit of software engineering techniques and hardware features of the facilities used. Modern compilers and IDE (Interactive Development Environments) can help the scientist in reducing the amount of technical knowledge and stuff required for obtaining efficient results from optimization operations.

I would like to aknowledge the organizers and teachers of the 13th edition of the Cineca Summer School on Parallel Computing, held at CINECA (Casalecchio di Reno) in july 2004, and the ones of the 2nd edition of the CASPUR Summer School on Advanced HPC, held at Castel Gandolfo, august-september 2006. I am grateful also to Dr. G. Perego of Aethia Power Computing Solutions S.r.l. and Dr. F. Conicella of the Bioindustry Park of Canavese (Colleretto Giacosa, Torino, Italy) for the support given in these years to the collaboration between the Dept. of Physics, Polytechnic of Torino, Prof. P.P. Delsanto's Research Group, and the Bioinformatics and High Performance Computing Laboratory of the Bioindustry Park of Canavese, collaboration through which I have been able to train myself in the field of HPC.

7. Bibliography

[Siemen2003a] S. Siemen, *Speed Compiler*, Linux Magazine **34** (July), 52-56 (2003)
[Siemen2003b] S. Siemen, *Go Faster*, Linux Magazine **34** (September), 66-70 (2003)
[Rebe2005] R. Rebe, *Compiler Rally*, Linux Magazine **59** (October), 50-52 (2005)
[Flynn1972] M. Flynn, *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput. C 21, 94 (1972)
[Migliardi2004] M. Migliardi, *Grid Computing: da dove viene e che cosa manca perchè diventi una realtà*, Mondo Digitale **??**, 21-31 (2004)
[Foster1999] I. Foster, C. Kasselman (editors), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, San Francisco (1999)
[Gough2005] B. Gough, *An Introduction to GCC for the GNU Compilers gcc and g++*, Network Theory Ltd., Bristol (2005) [Available in HTML format at http://www.network-theory.co.uk/gcc/intro/]
[Stallman2006] R.M Stallman and the GCC Developer Community, *Using the GNU Compiler Collection (GCC)*, GNU Press (Free Software Foundation), Boston, 2006.
[Briant2003] R.E. Briant, D. O'Hallaron, *Computer Systems. A Programmer's Perspective*, Prentice Hall, 2003.
[Patterson2005] D. Patterson, J. Hennessy, *Computer Organization and Design*, MK Publisher, 3rd edtion, 2005.
[Dowd1998] K. Dowd, C. Severance, *High Performance Computing*, O'Reilly, 2nd edition, 1998.

8. List of useful Web sites

- www.top500.org : the Web site with the list of the first 500 most performant supercomputers of the world, compiled every six months by a committee representing different institutions and industrial firms. The classification is made by running on each supercomputer a specific code for solving a linear algebra problem making use of the library .....
- www.beowulf.org : the official Web site of the communities promoting projects involving Beowulf-like Linux clusters.
- www.ibm.com/servers/eserver/pseries/library/sp_books : IBM Web site dedicated to the documentation about a class of its famous suercomputers, the E-Servers SP clusters.
- www-unix.mcs.anl.gov/mpi/index.html : MPI official Web site hosted by the Mathematics and Computer Science division of the Argonne National Laboratory, USA.
- www.mpi-forum.org/ : MPI Forum official Web site.

- www.openmp.org/ : OpenMP official Web site.
- www-unix.mcs.anl.gov/mpi/mpich/ : MPICH, one of the most famous and surely the most portable MPI implementation.
- www.lam-mpi.org/ : official site of the MPI-LAM implementation of MPI by a common initiative of Indiana University, University of Notre Dame (USA), Ohio State University
- www.open-mpi.org/ : official site of OpenMPI implementation of MPI
- www.intel.com/software/products/compilers/linux/ : official site of the Intel C/C++ compilers for Linux platforms, where a bucnh of useful tutorials, guides and datasheets can be found about those compilers, their use on Linux platforms and in general on optimization of serial and parallel codes running on Intel-based architectures.
- www.intel.com/support/performancetools/c/linux/ : other useful and interesting site from Intel about its C/C++ and Fortran compilers for Linux platforms. Intel has been promoting the diffusion of its CPUs and dedicated software (compilers, libraries, debugging and profiling tools, tuning and optimizing toolboxes, etc. ...) within the academic and industrial HPC communities, not only offering its compilers for Linux platforms free according to a non-commercial license free of charge (only for private use, not for academic or teaching use) but also supplying them with a huge amount of free documentation, which is very useful in order to learn optimization techniques (for Intel platforms, obviously). AMD has not yet adopted such a policy of public/free-of-charge support to HPC users who have facilities relying on AMD CPUs. There is also a project by Intel devoted to the compilation of the entire Linux kernel (version 2.6.x) using its compilers, in order to show the increase of performance that can be obtained using its own compilers on Linux platforms. Have look at the URL www.intel.com/support/performancetools/c/linux/kernel.htm

- www3.intel.com/cd/software/products/asmo-na/eng/download/download/index.htm : Intel has released since the beginning of 2006 also a bunch of software dedicated to optimization and optimal use of codes on its platforms, e.g. the Intel Math Kernel Library (Intel MKL), which is the most optimized library of scientifi computing functions/subroutines for Intel-based platforms, the Intel VTune Performance Analyzer, the Intel Threading Building Blocks for Linux, a toolbox dedicated to multi-threading programming on Intel single-core, multi-core architectures or Intel-based SMPs.
- www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/index.htm : Intel knowledge base on multi-threading programming, for single and multi-core CPUs.
- www.coyotegulch.com/ : interesting site about performance analysis of codes compiled with different compilers.
- http://gcc.gnu.org/ : the official Web site of the GNU Compiler Collection (GCC) initiative and software.