TUTORIAL - 3B - HPC INFRASTRUCTURES

part 1

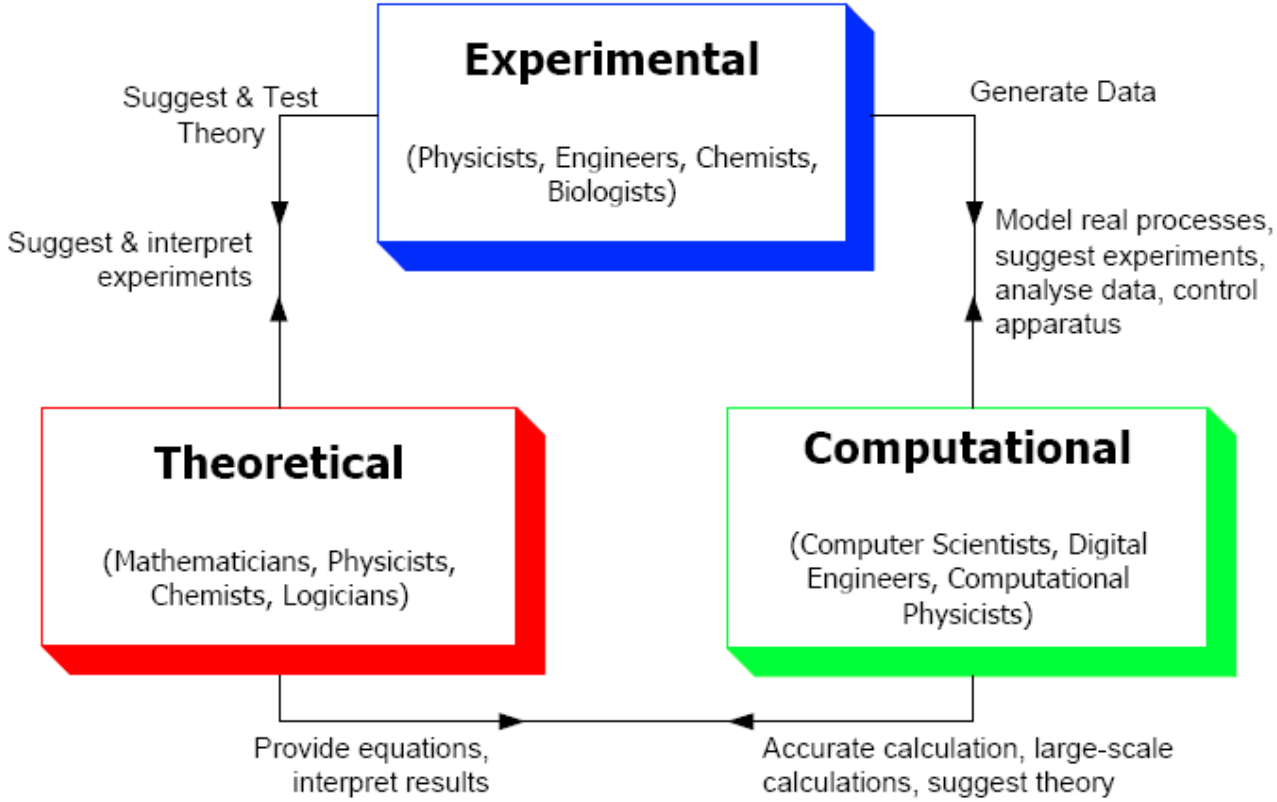
Claudio Arlandini: CILEA System Administration Group
Raffaele Ponzini: CILEA HPC Group

Index of the presentation

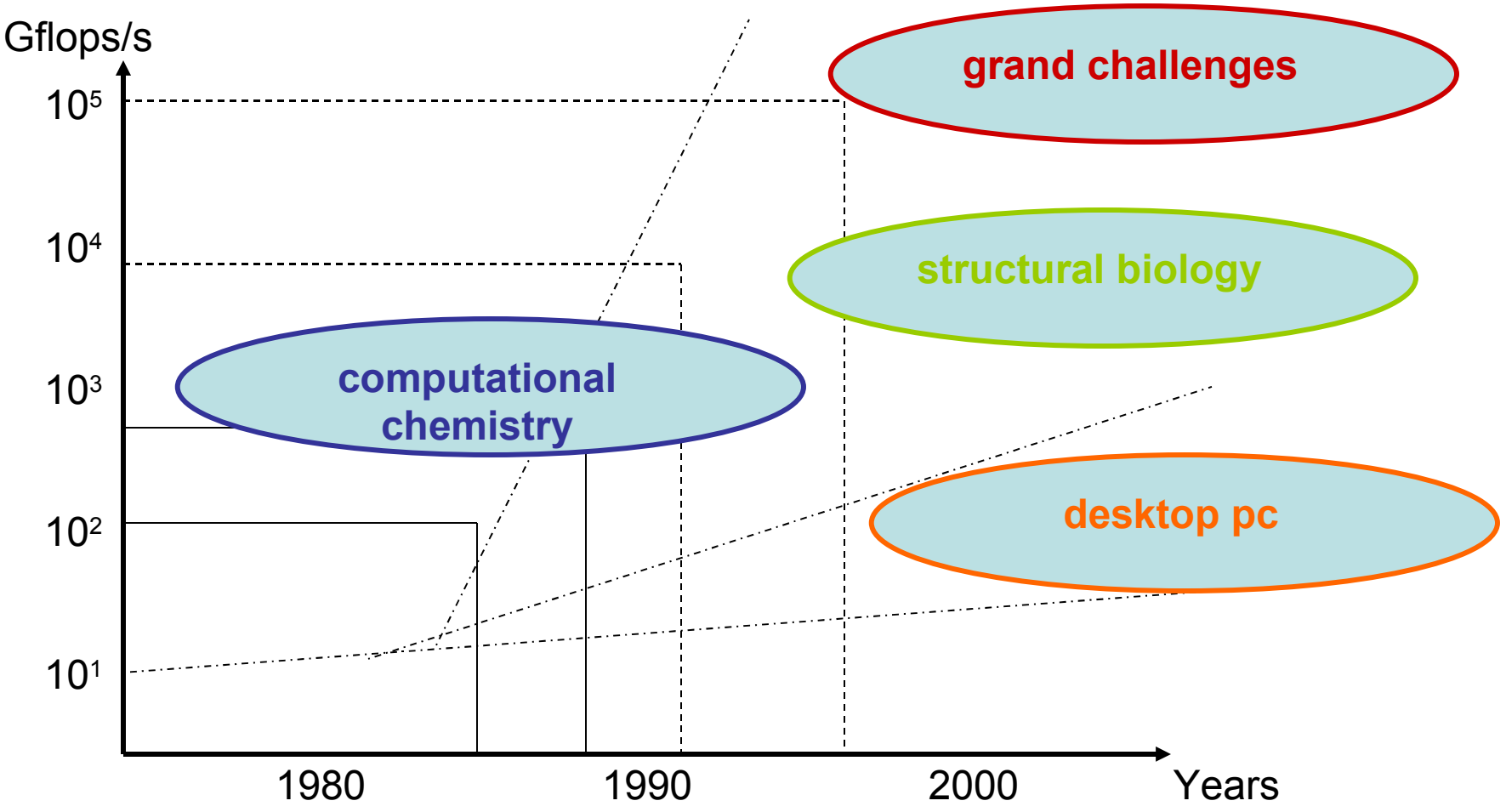
- ✓ Perspectives and requirements of bioinformatics in the Grid era
- ✓ Flynn classification of architectures
- ✓ Memory classification of architectures
- ✓ Topology of interconnection networks
- ✓ Technologies of interconnection networks
- ✓ Architectures hosted by CILEA

Perspectives and requirements in bioinformatics in the Grid era

- ✓ Grand challenges in informatics
- ✓ Top ten challenges for bioinformatics



Grand challenges in informatics



computational costs of about 10^6 – 10^{13} Gflops/s

Today's top 500

Rank	Site	Computer	Proc.	Y	$R_{max(*)}$	$R_{peak(*)}$
1	DOE/NNSA/LLNL	BlueGene/L	131072	'05	$28*10^4$	$36*10^4$
2	IBM Thomas J. Watson Research Center	BlueGene/L	40960	'05	$91*10^3$	$11*10^4$
3	DOE/NNSA/LLNL	ASC Purple	10240	'05	$63*10^3$	$11*10^4$

[(*)Gflops/s]

Computational modeling of biochemical systems—the application of high-speed computing technology to simulate and visualize complex, integrated biological processes

Bioinformatics—databasing, networking, and analysis of biological data

Bioinstrumentation—the application of physical and engineering technologies to laboratory automation, robotics, medical device development, and healthcare.

Top ten challenges for bioinformatics

Source: Ewan Birney, Chris Burge, Jim Fickett

[1] Precise models of where and when transcription will occur in a genome (initiation and termination)

[2] Precise, predictive models of alternative RNA splicing

[3] Precise models of signal transduction pathways; ability to predict cellular responses to external stimuli

[4] Determining DNA-protein, RNA-protein, protein recognition codes

[5] Accurate *ab initio* protein structure prediction

[6] Rational design of small molecule inhibitors of proteins

[7] Mechanistic understanding of protein evolution

[8] Mechanistic understanding of speciation

[9] Development of effective gene ontologies:
systematic ways to describe gene and protein function

[10] Education: development of bioinformatics curricula

Flynn classification of architectures

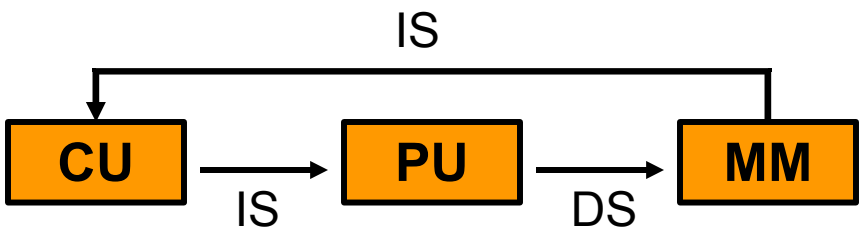
- Flynn classification gives the taxonomy of HPC architectures by means of the multiplicity of the hardware used to manage the stream.
- In this way we can distinguish architectures as follow:

3. SISD: Single Instruction Single Data Stream
4. SIMD: Single Instruction Multiple Data Stream
5. MISD: Multiple Instruction Single Data Stream
6. MIMD: Multiple Instruction Multiple Data Stream

SISD

Single Instruction stream over a Single Data stream

In this architecture configuration (wich correspond to the Von Neumann machine schema, and actually to all the desktop PC that we use) a single processor executes a single instruction stream. Data are stored in a single memory.



LEGENDA:

CU: Control Unit

PU: Processing Unit

IS: Instruction Stream

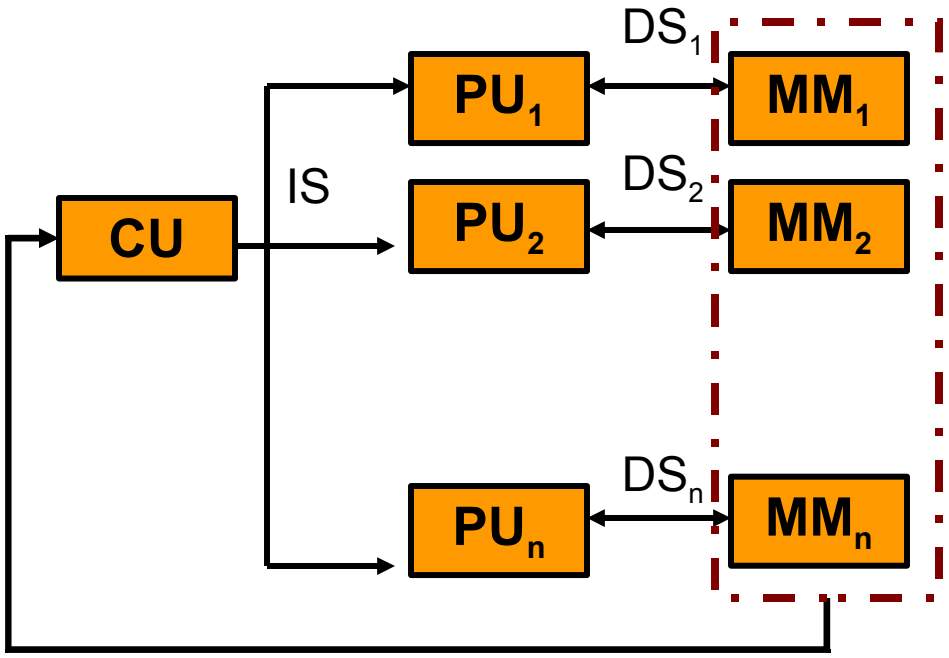
DS: Data Stream

MM: Main Memory

SIMD

Single Instruction stream over Multiple Data stream

In this architecture configuration a single processor executes a multiple instruction stream. The parallel computational model is the so called *synchronous* (in the sense that all the processors work in lock-step) and the processor is in general referred as a *vector* or *array* processor.



LEGENDA:

CU: Control Unit

PU: Processing Unit

IS: Instruction Stream

DS: Data Stream

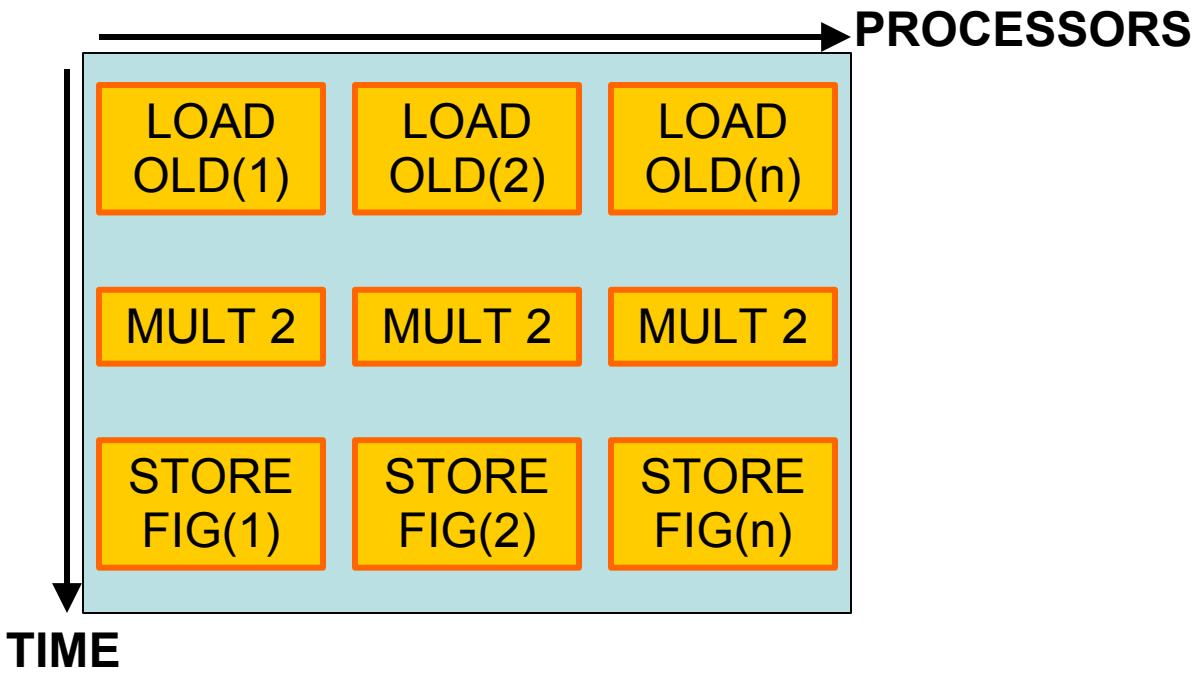
MM: Main Memory

- In order to take an advantage from such an architecture the application should make the same operation over a large number of data points.
- In this architecture such an operation can be done at once.
- This kind of instruction are called “vectorized” because they threat the data set as they are stored in vector.

Example

Multiply by 2 the value of the brightness in all the pixels of an image:

```
FIG(i) = OLD(i)*2
==> LOAD OLD(i),MULT 2, STORE FIG(i)
```



Vectorial machines

- Cray C90, Cray T90

  [http://www.cray-cyber.org/systems/c90.php]

  [http://www.top500.org/orsc/1998/crayv.html]
- Nec SX-6

  [http://www.sw.nec.co.jp/hpc/sx-e/sx6/specification_e.html]
- APE 1000, APE NEXT

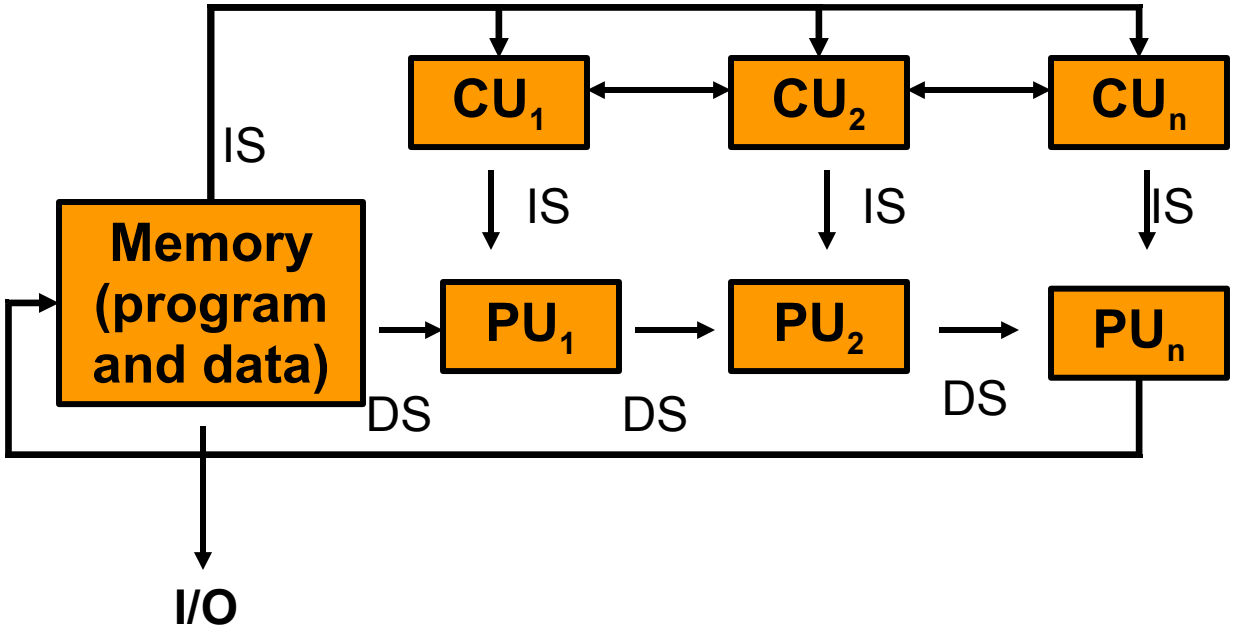
  [http://www.eurotech.com/IT/innovazione.aspx?pg=apenext]

MISD

Multiple Instruction stream over Single Data stream

In this architecture configuration many processors performs different operations on the same data. This parallel computational model is not very diffuse and only few implementations exists.

Practical applications that can take advantage from this architecture are redundant systems where several backup systems are needed in case one fails.



LEGENDA:

CU: Control Unit

PU: Processing Unit

IS: Instruction Stream

DS: Data Stream

MM: Main Memory

I/O: Input Output

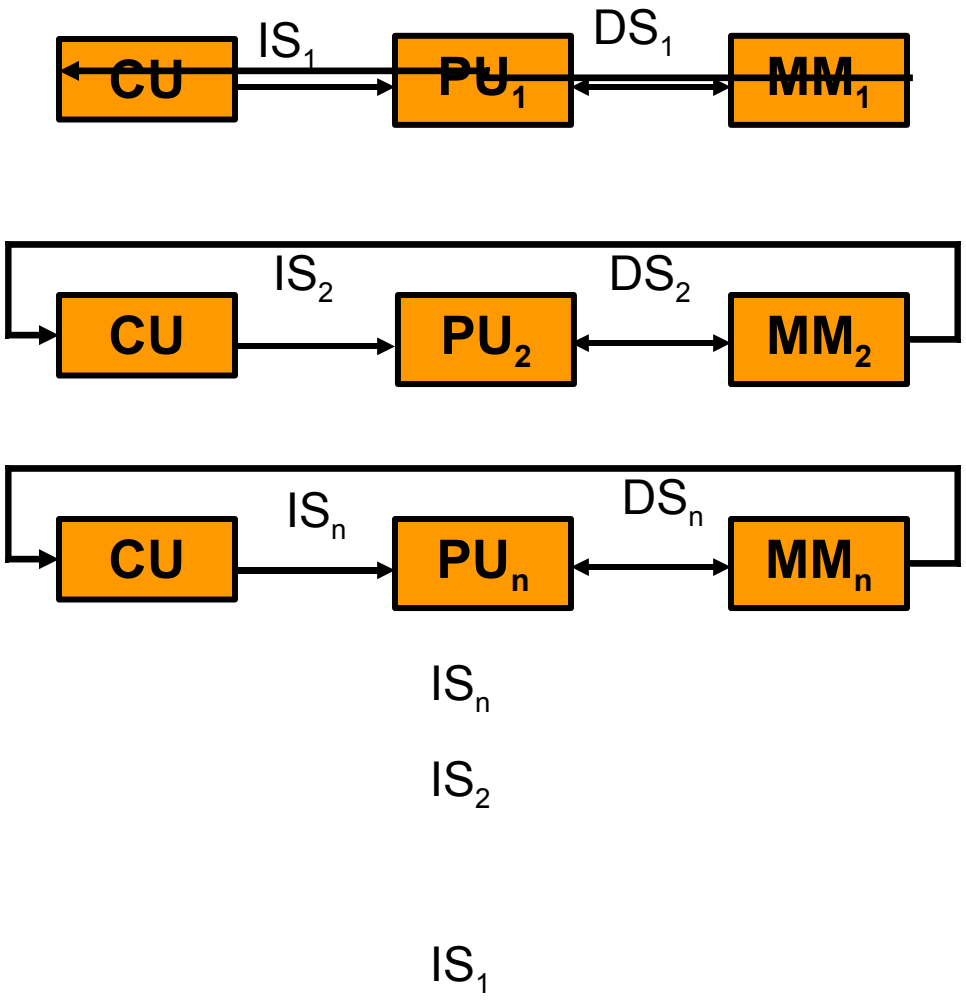
MIMD

Multiple Instruction stream over Multiple Data stream

In this architecture configuration different processors performs different operations on the shared or different data. This parallel computational model can be interpreted as an evolution of the SISD model.

The MIMD model can be divided in other sub-categories: *shared memory* (so called multiprocessor) and *distributed memory* (so called cluster).

Distributed memory (cluster)



LEGENDA:

CU: Control Unit

PU: Processing Unit

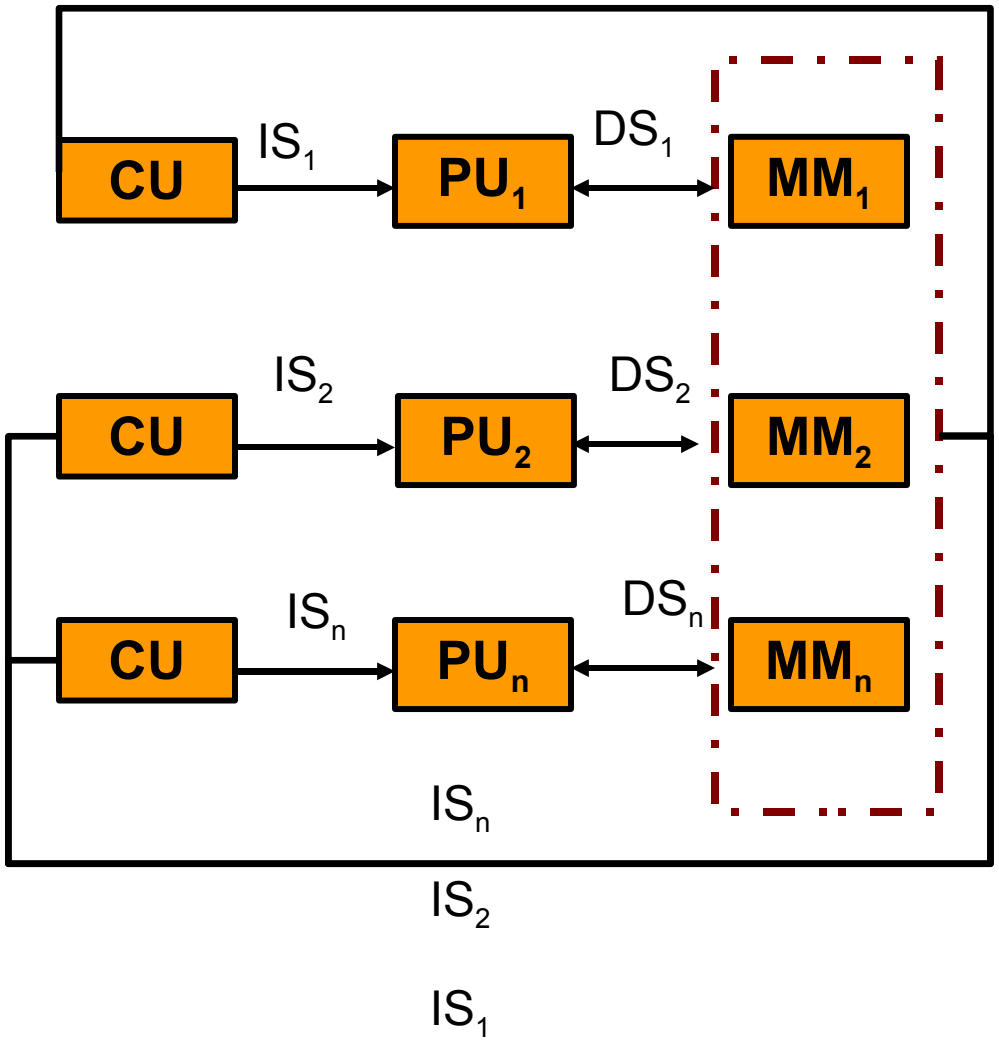
IS: Instruction Stream

DS: Data Stream

MM: Main Memory

I/O: Input Output

Shared memory (multiprocessor)



LEGENDA:

CU: Control Unit

PU: Processing Unit

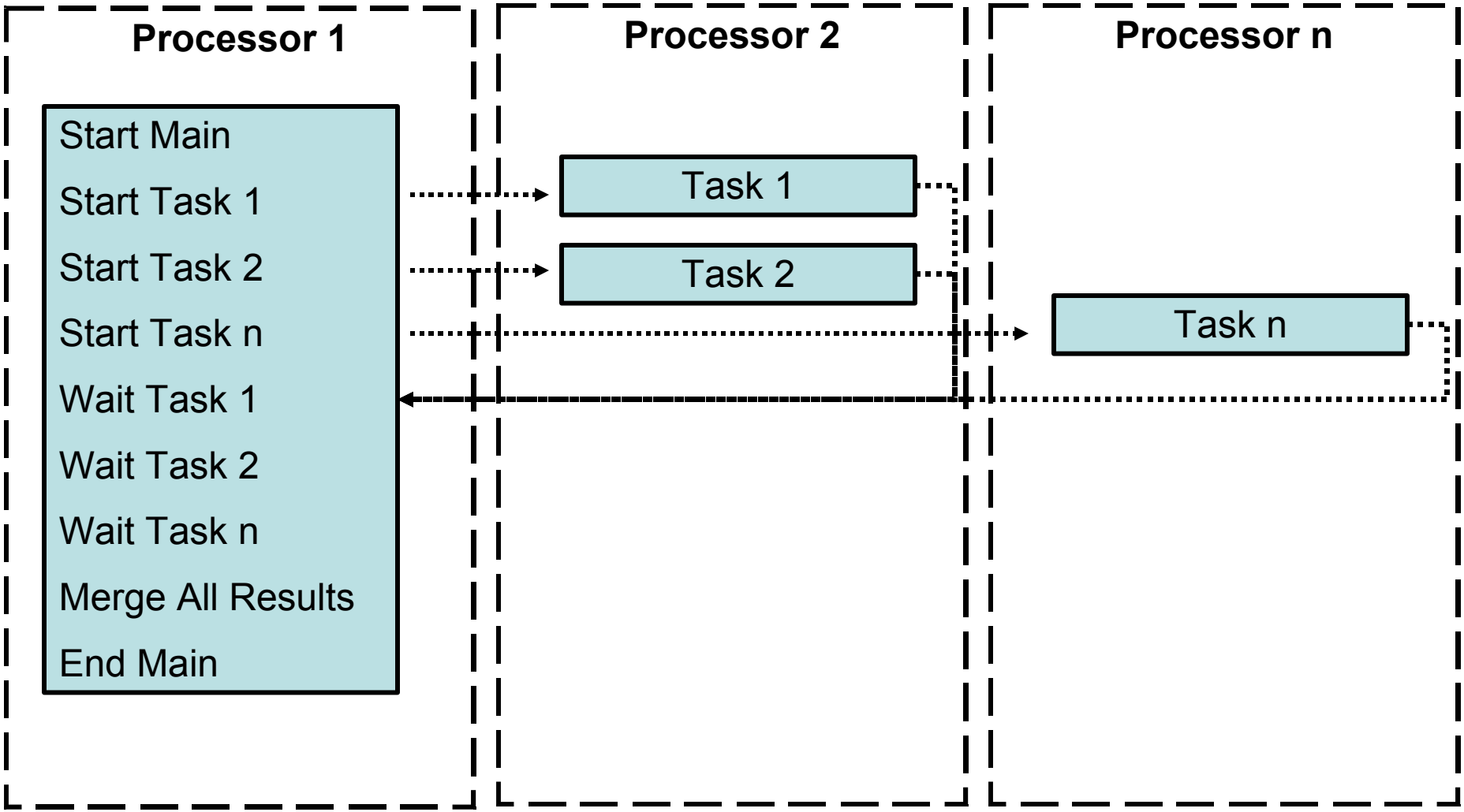
IS: Instruction Stream

DS: Data Stream

MM: Main Memory

I/O: Input Output

Example


Processor 1
Processor 2
Processor n
Start Main
Start Task 1
Start Task 2
Start Task n
Wait Task 1
Wait Task 2
Wait Task n
Merge All Results
End Main
Task 1
Task 2
Task n

Cluster computing

Nowadays the most implemented kind of architecture is a MIMD mixed model of shared (inside the single node) and distributed (between the different nodes) memory use.

In modern clusters, processors are grouped in nodes where they share locally the memory. Globally different nodes communicate by means of an interconnection network.

This kind of architecture finds his historical reason in the costs/benefits trend of hardware and network technology

Clusters

- BlueGene/L IBM

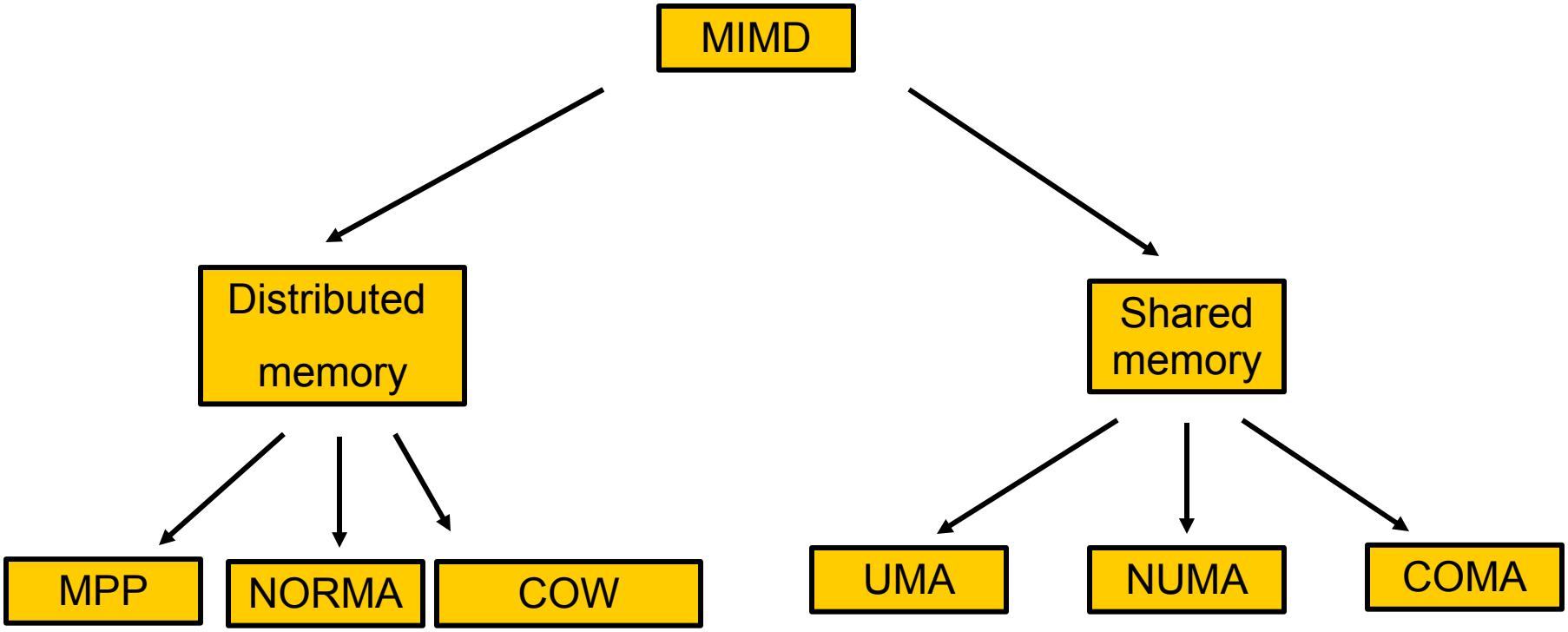
  [http://it.wikipedia.org/wiki/Blue_Gene#Blue_Gene.2FL]
- Mare Nostrum IBM

  [http://www.bsc.es/index.html]
- Cray X1

  [http://www.cray.com/products/x1e/]
- ASCI Purple

  [http://www.llnl.gov/asci/platforms/purple/]

Memory classification of architectures



Memory classification of architectures

- Beside the Flynn's classification there is another one based on the distribution of the memory.
- In this classification we distinguish between **shared** and **distributed** memory and inside these two categories:

-MPP model: Massively Parallel Processing

-COW model: Cluster Of Workstations

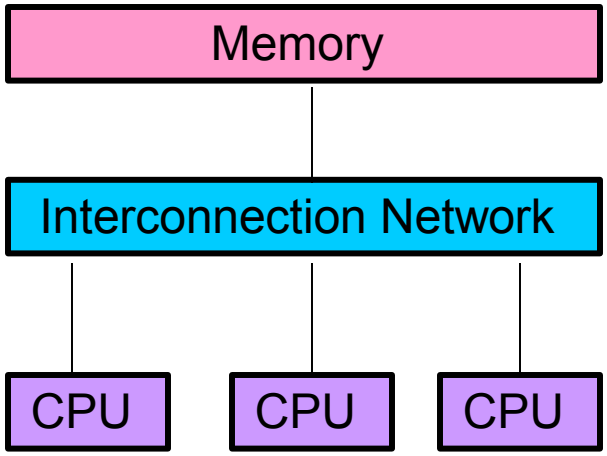
-UMA model: Uniform Memory Access

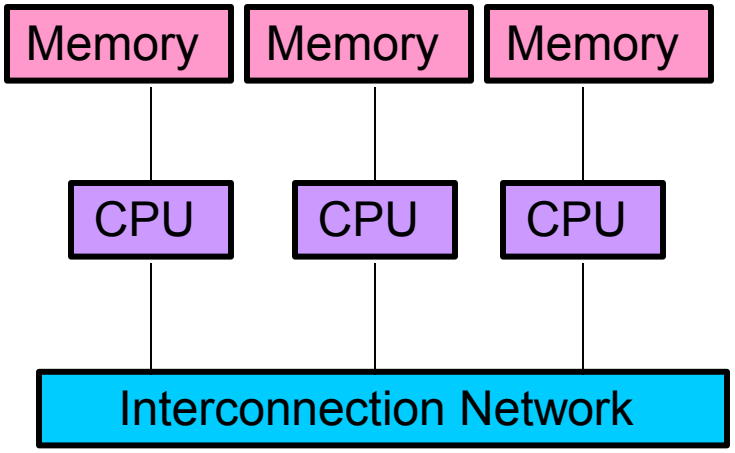
-NUMA model: Non Uniform Memory Access

-NORMA Model: NO-Remote Memory Access

-COMA model: Cache Only Memory Access

Shared/Distributed

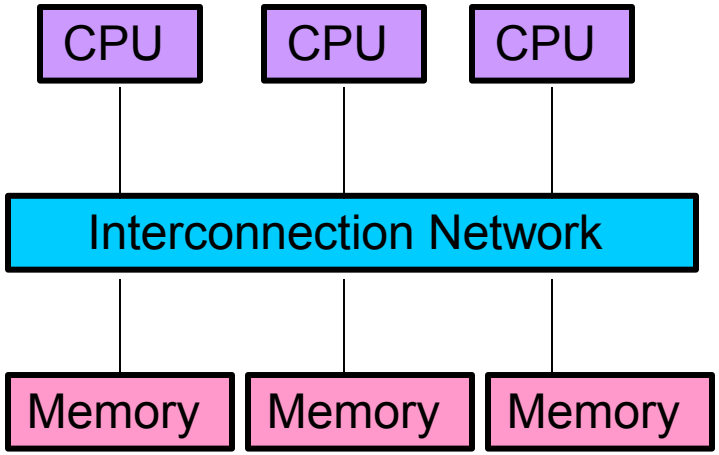




- **Shared:** all the CPU access the same space. In this kind of system memory can became a bottleneck. In general these systems are constituted by a limited number of processors.
- **Distributed:** every CPU uses a local memory. CPU communicate by means of messages. In general these systems are constituted by a large number of processors (nodes).

UMA

Uniform Memory Access



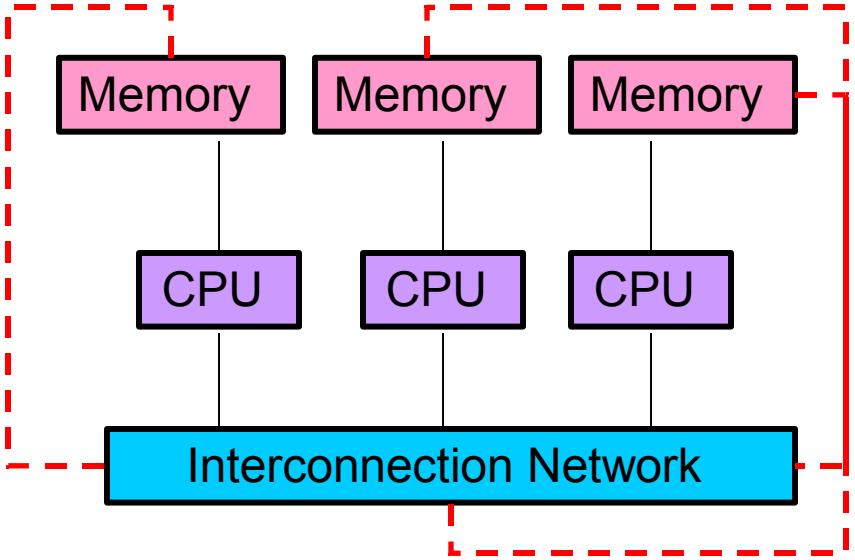
Memory is shared equally by all the processors.

This model is easily expandable and easy to program, the use of common variables permit to synchronize the processors.

The access to the I/O devices can be symmetric (all the processors can access the devices) or anti-symmetric (only some processors can access the devices)

NUMA

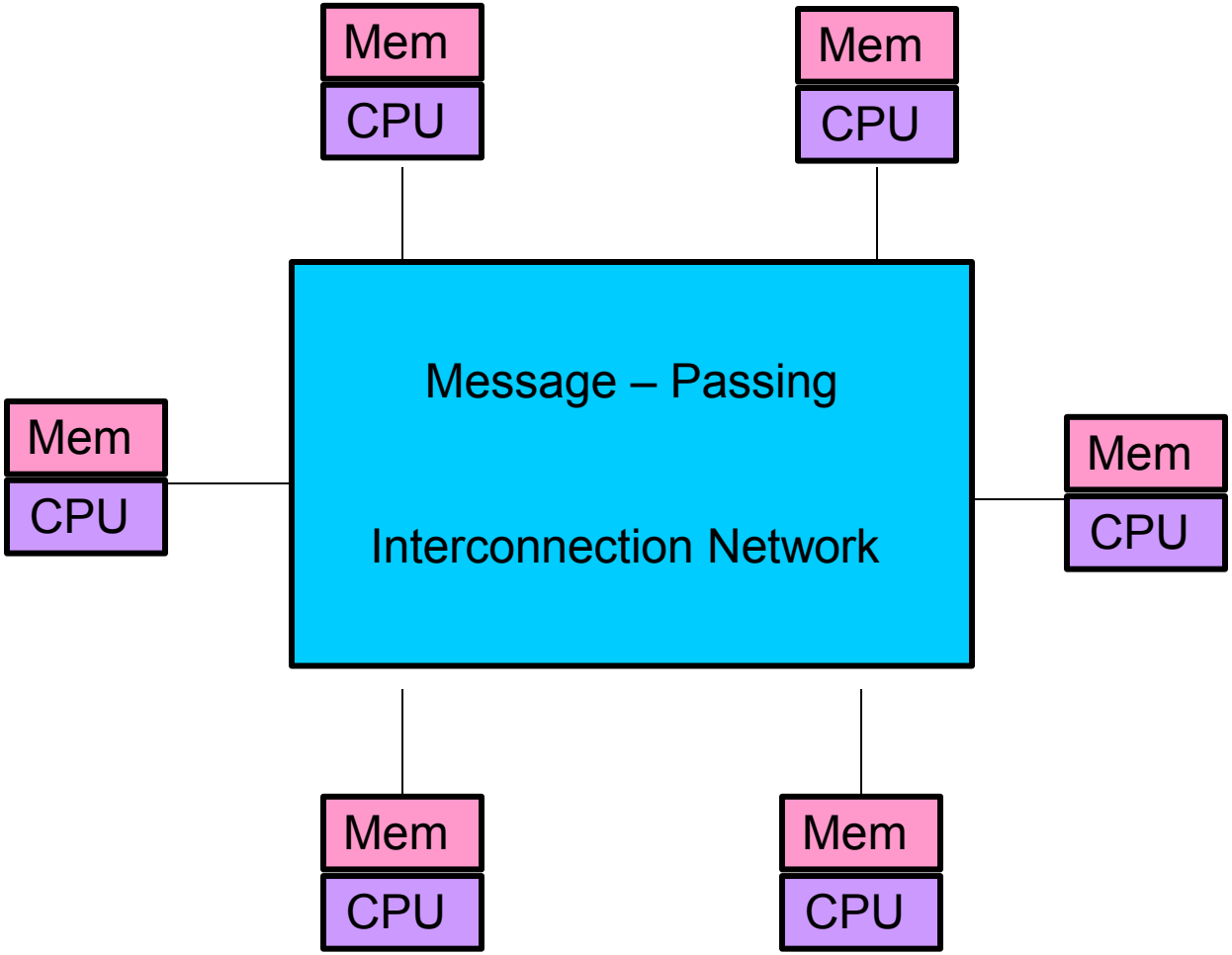
Non Uniform Memory Access



Each processor has its own memory but can access the memory of another processor by means of the interconnection network.

NORMA

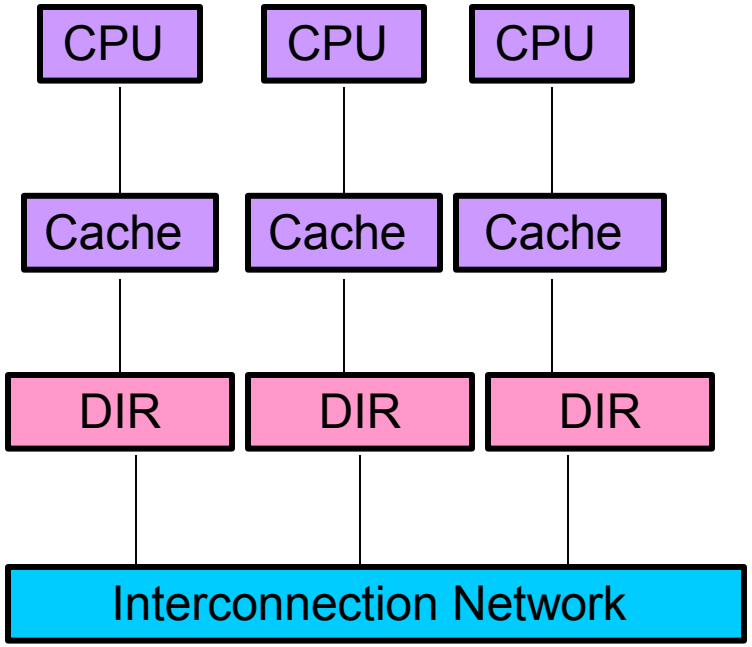
NO-Remote Memory Access



Each node has a private memory and communicate with other processors only by means of a message passing paradigm. These systems are more difficult to program because of the need of specifing the synchronization between processors and the data distribution.

COMA

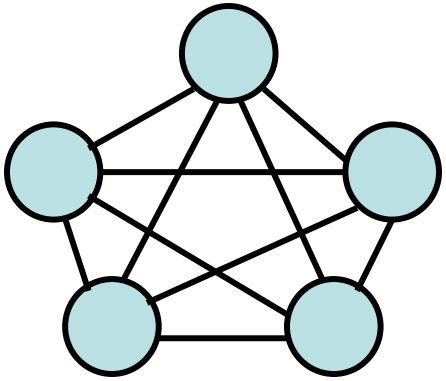
Cache Only Memory Access



This is a particular configuration of simple NUMA system where the single processor can access only a private cache which communicates with a private memory.

Topology of interconnection networks

In general having N computing nodes the complexity of a fully connected network is proportional to $O(N^2)$.



In order to afford the cost (in term of complexity) of a interconnection network in a cluster several configuration have been suggested:

-Mesh

-Fat Tree

-Shuffle-Exchange

-Omega

-Butterfly

-Hypercube

Definitions

Networks are usually represented as graph where:

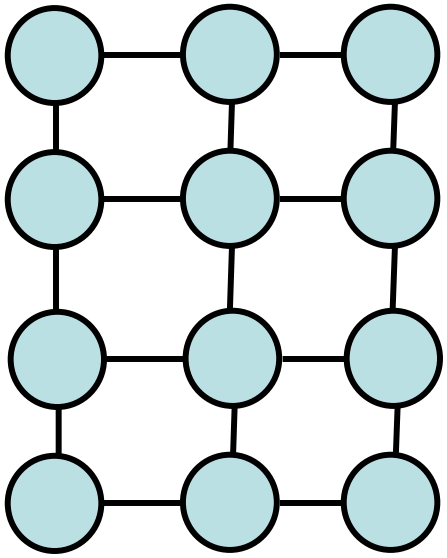
- **Vertices** corresponds to single processor node
- **Edges** corresponds to communications links and they can be unidirectional or bidirectional

Important parameters of an interconnection network include:

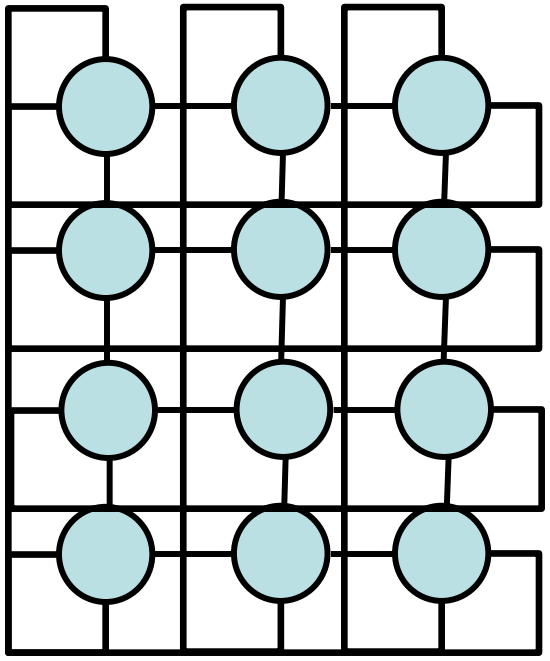
- 1. **Network Diameter** – The longest of the shortest paths between various pairs of nodes, which should be relatively small, is the latency is to be minimized. The network diameter is more important with store-and-forward routing (when a message is stored in its entirety and retransmitted by intermediate nodes) than with wormhole routing (when a message is quickly relayed through a node in small pieces).
- 2. **Bisection (band)width** – The smallest number (total capacity) of links that need to be cut in order to divide the network into two sub networks of half the size. This is important when nodes communicate with each other in a random fashion. A small bisection (band)width limits the rate of data transfer between the two halves of the network, thus affecting the performance of communications intensive algorithms.
- 3. **Vertex or node Degree** – The number of communications ports required of each node, which should be a constant, independent of network size if the architecture is to be readily scalable to larger seizes. The node degree has a direct effect on the cost of each node, with the effect being more significant for parallel ports containing several wires or when the node is required to communicate over all ports at once.

Mesh

mesh



torus mesh

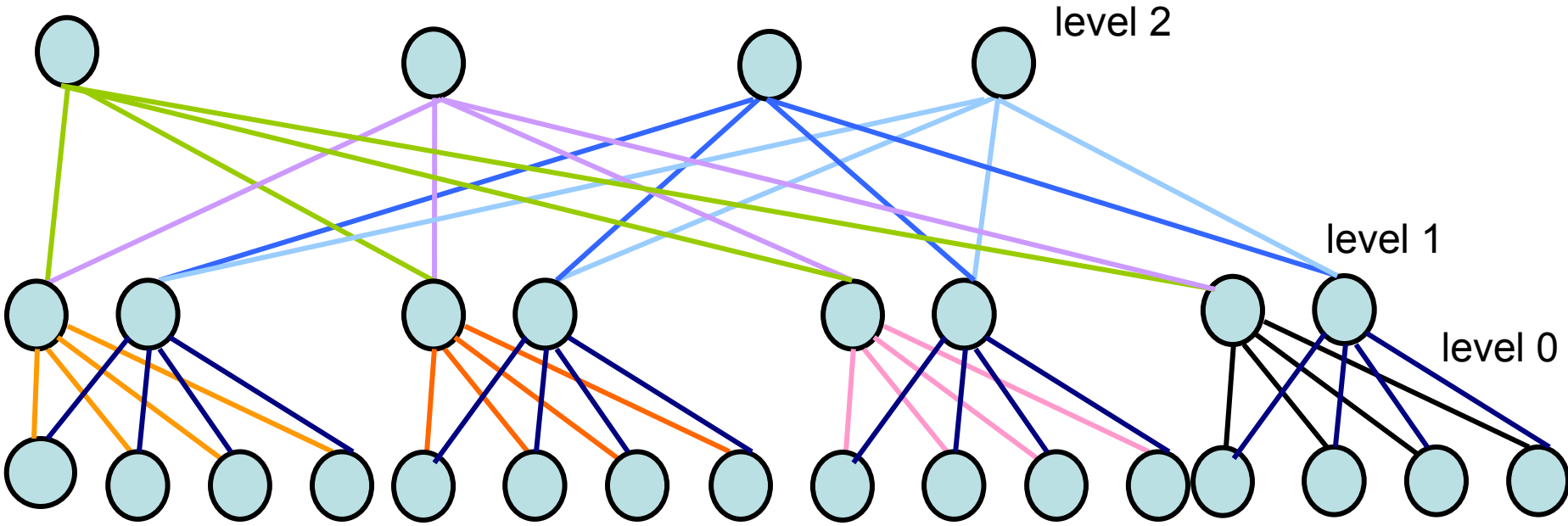


In this network configuration nodes are disposed along a q-dimesional mesh.

The communication is permitted only between adjacent nodes.

An alternative mesh configuration is the torus mesh where the communication is permitted also between some non-adjacent nodes.

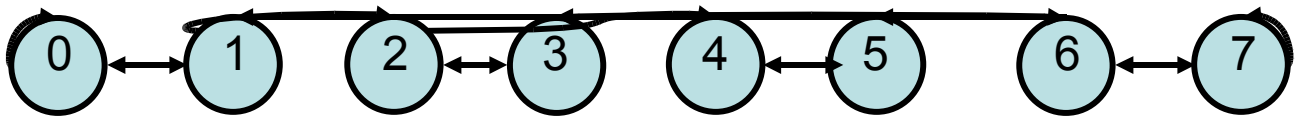
Fat Tree



A fat tree network have this characteristic: having `n` nodes the number of levels is done by `log(4*n)` [every node have 4 connection with super-level and 2 connection with the sub-level]

NOTE: *avogadro.cilea.it* has a fat tree network myrinet connection linking 128 Xeon 3GHz nodes

Shuffle-Exchange

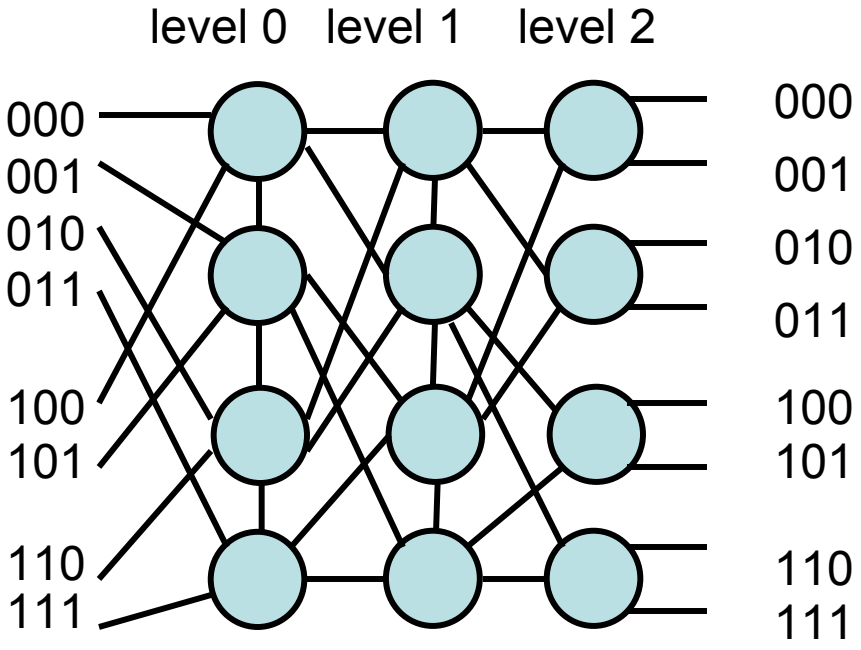


Having n nodes [`numbered from 0 to n-1`] the interconnection is done by means of two family:

-shuffle: connect the node `i-th` with the `2*i*mod(n-1)-th`

-exchange: connect couples of nodes that differs only for their less significant bit

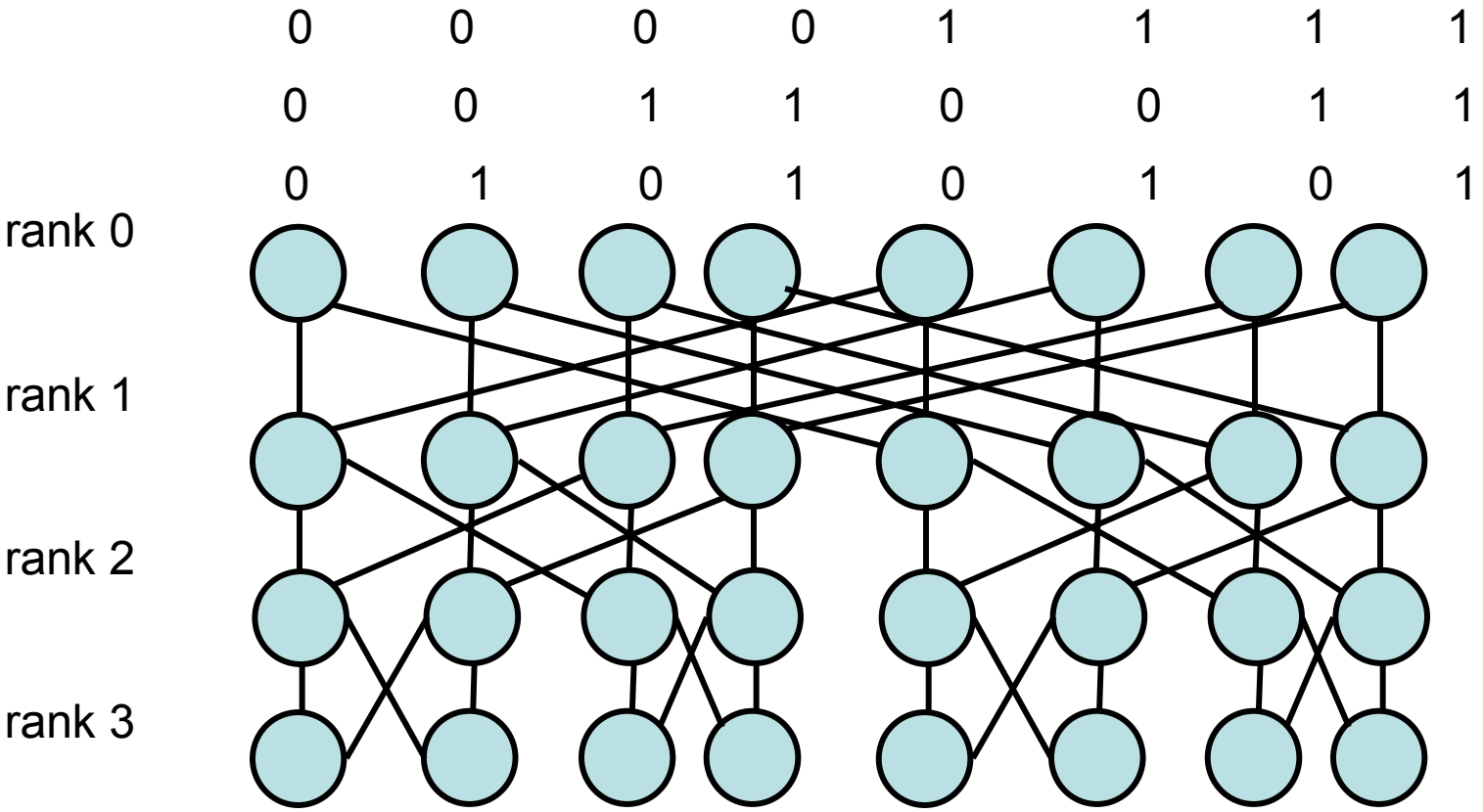
Omega



An `n x n` omega network is made by `log(2*n)` levels.

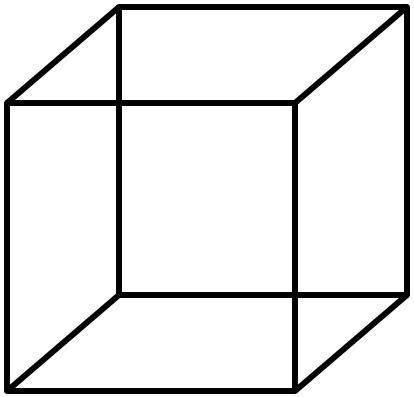
Every level is made by `n/2` switches.

Butterfly

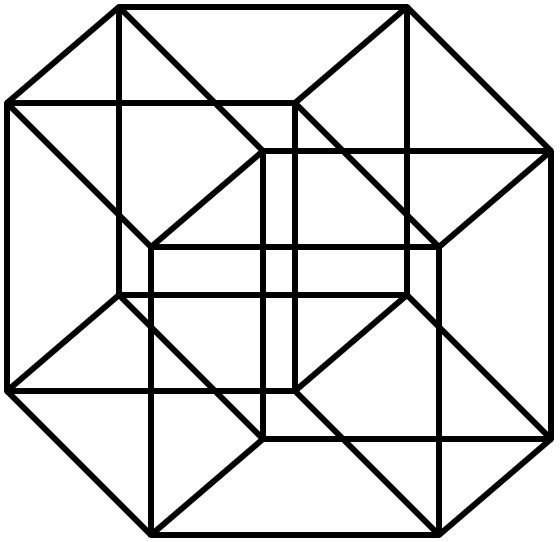


A butterfly network is made by $(n+1)*2^n$ nodes divided in n+1 ranks each containing 2^n nodes.

Hypercube



Hypercube 3D



Hypercube 4D

An hypercube network is made by $n=2^k$ nodes disposed along the vertices of an k-dimensional cube.

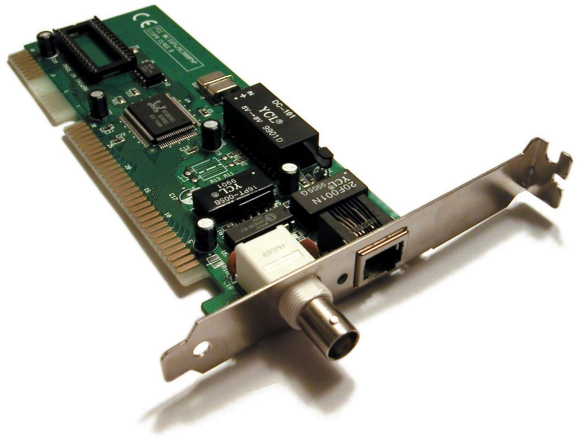
Summary

Network topology	# of Nodes	Network diameter	Bisection width
1D Mesh	K	K-1	1
1D Torus	K	k/2	2
2D Mesh	K^2	2k-2	K
2D Torus	K^2	K	2k
3D Mesh	K^3	3k-3	K^2
3D Torus	K^3	3k/2	2 K^2
Butterfly	$2^k(k+1)$	2k	2^k
Hypercube	2^k	k	2^{k-1}
Shuffle Exchange	2^k	2k-1	$>2^{k-1}/k$

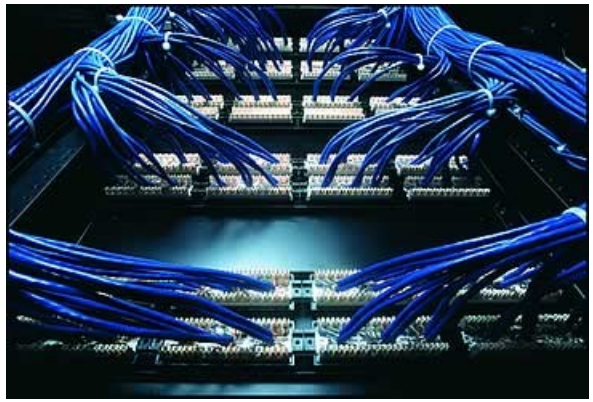
Technologies of interconnection networks

- ✓ Ethernet
- ✓ Myrinet
- ✓ Infiniband

Ethernet



The name coming from the physical concept of the ether, **Ethernet** is a large, diverse and general family of frame-based network technologies for LAN (Local Area Networking). Ethernet has been standardized as IEEE 802.3. During the ’90s became the most popular LAN technology replacing competing LAN standards such as Coaxial-cable Ethernet, FDDI, and others.
From the intensive computing point of view they suffer from single points of failure and from bandwidth choke points where a lot of traffic is forced down a single link.



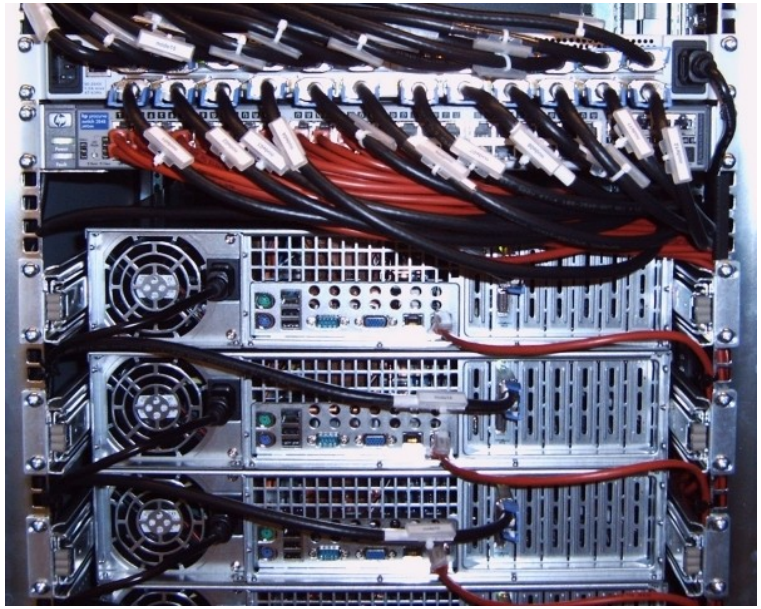
Myrinet

Myrinet, ANSI/VITA 26-1998, is a high-speed LAN system designed by Myricom to be used as an interconnect between multiple machines to form computational clusters with much less protocol overhead than standard Ethernet and better throughput, less interference, and less latency while using the host CPU.
Myrinet is often used directly by programs that "know" about it, thereby bypassing a call into the OS in order to use these feature.
For supercomputing, the low latency of Myrinet is very important (more than its throughput performance) because high-performance parallel system tends to be bottlenecked by its slowest sequential process, which is often the latency of transmission of messages across the network.
In the November 2005 TOP500, the number of supercomputers using Myrinet is down to 101 computers, or 20.2%.

Infiniband

InfiniBand is a high-speed serial computer bus, intended for both internal and external connections mantained by the InfiniBand Trade Association (IBTA).

InfiniBand is having some success against other high performance computing switch fabric vendors, notably Quadrics and Myricom (Myrinet). In the supercomputing segment, Infiniband's closest competition remains the low cost and relative ubiquity of gigabit ethernet.. As gigabit Ethernet evolves toward 10-gigabit Ethernet, InfiniBand will face stiffer competition. Infiniband will retain a higher maximum throughput (on QDR hardware) overall, but at the 10 Gbit/s and below level, the primary advantage of Infiniband becomes its fabric architecture (rather than its speed).



Architectures hosted by CILEA

- ✓ **Xeon** processors @ Avogadro.cilea.it
- ✓ **Opteron** processors @ Golgi.cilea.it &
  @ Michelangelo.cilea.it

Xeon processors @ Avogadro.cilea.it

AVOGADRO

NODES: 128 SuperMicro - chassis Super SC811i-350

CPU: 256 Intel Xeon 3.06 GHz

RAM: 2 GB per CPU

STORAGE: 6400 GB; 20GB local node scratch; 1TB of shared scratch

NETWORK INTERFACE: Myrinet and Gigabit Ethernet

Opteron processors

@ Golgi.cilea.it

GOLGI

NODES: 55 (18 + 37)

CPU: 220 (72 AMD Opteron 848 2.2 GHz; single-core / 74 AMD Opteron 275 2.2 GHz; dual-core)

RAM: 2-4 GB per CPU

STORAGE: 2.6 TB

NETWORK INTERFACE: Infiniband 4x and Gigabit Ethernet

Opteron processors

@ Michelangelo.cilea.it

MICHELANGELO

NODES: 70 (Blades)

CPU: 280 (140 AMD Opteron 275 2.2 GHz; dual-core)

RAM: 4 GB per CPU

STORAGE: 25 TB

NETWORK INTERFACE: Infiniband 4x and Gigabit Ethernet

Performance issues: how to get the best from your x86, x86_64 or em64t system?

1. Know your processor
2. Know your compiler
3. Know your application

Performance issues:
how to get the best from your x86, x86_64 or em64t system?

1. **Know your processor**
2. Know your compiler
3. Know your application

1. Know your processor

Shortly: we will speak only of

Xeon	Intel	32bit
EM64T	Intel	64bit
Opteron	AMD	64bit

1. Know your processor

Xeon was introduced in 2001 as Pentium4 version for workstations.

In 2002 a 130 nm version of the Xeon (Prestonia) was released, supporting Intel's new Hyper-Threading technology and having a 512 KiB L2 cache.

Hyperthreading allows multiple threads to run simultaneously [certain sections of the processor, those that store the architectural state, are duplicated, but not the main execution resources – this allows the processor to pretend being two "logical" processors]

It supports SSE2 instruction set

SSE adds eight 128-bit registers, enabling the programmer to perform math of any type using entirely these registers. This increases the perforrmance, since FPU and SSE instructions may be executed on the same clock cycle.

1. Know your processor

Opteron was released in 2003 as the first processor to:

- allow native execution of x86 32bit code without speed penalties
- allow native execution of 64bit applications

May handle 2^{40} bytes of memory access (1TB) and 2^{48} bytes of virtual address space

In a multi-processor system the CPUs communicate through high speed HyperTransport links (better scaling).

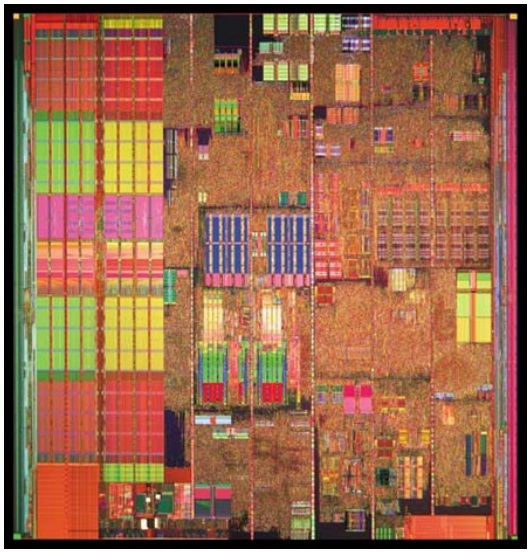
In May 2005 it was introduced the first dual-core CPU.

Each chip contains two separate processor cores sharing the same cache. The result Is almost doubling the performance with almost the same electrical consumption.

It supports 3Dnow! instruction set

Also a set of fast math instructions. They are faster than SSE, but do not use special registers, so FPU and 3Dnow! Instructions cannot be executed simultaneously.

1. Know your processor



EM64T is Intel implementation of the 64bit
Extension of the 32bit architecture, released in 2005.

Differences with Opteron:

- Opteron supports 3Dnow! Instruction set (and now a subset of SSE3)
- EM64T supports SSE3 Instruction set
- Other instructions are different – this may be reflected in application performances

First EM64T addressed only 2^{36} bytes of memory and were said to have stability issues.

The first dual core Intel processor was recently announced.

Performance issues:
how to get the best from your x86, x86_64 or em64t system?

1. Know your processor
2. **Know your compiler**
3. Know your application

2. Know your compiler

Many C/C++ and Fortran compilers do exist on the market.
Let's focus briefly on three:

- GNU: it's free, it's good and stable, but not particularly optimized
- INTEL: it's **not** free, and optimized for INTEL (obviously) processors
- Portland: it's **not** free, but you should think about it if you want to obtain the maximum from your Opteron.

2. Know your compiler

Be aware of commercial (mis)practices:

there is nothing *in principle* that makes Intel compilers unsuitable for Opterons
BUT reverse engineering shows that the Intel compiler check if the CPU is not
a genuine Intel and in that case it disables the use of many important optimizations
that might be applied, like the use of SSE3 instructions.

See e.g.: http://www.swallowtail.org/naughty-intel.html

2. Know your compiler

In the following slides we will show some of the most important optimization options for the three before mentioned compilers.

BE AWARE: no universal recipes exist!!!

Each application is different, therefore for each application you should make a complete test to discover the set of optimization options that give you the best possible results.

Many, many other options exist for each compiler,
STUDY CAREFULLY YOUR MANUAL!!

2. Know your compiler

GNU compiler important optimization options (1/2):

-O0 or no -O option (default)

At this optimization level GCC does not perform any optimization and compiles the source code in the most straightforward way possible. Each command in the source code is converted directly to the corresponding instructions in the executable file, without rearrangement. This is the best option to use when debugging a program and is the default if no optimization level option is specified.

-O1 or -O

This level turns on the most common forms of optimization that do not require any speed-space tradeoffs. With this option the resulting executables should be smaller and faster than with -O0. The more expensive optimizations, such as instruction scheduling, are not used at this level. Compiling with the option -O1 can often take less time than compiling with -O0, due to the reduced amounts of data that need to be processed after simple optimizations.

-O2

This option turns on further optimizations, in addition to those used by -O1. These additional optimizations include instruction scheduling. Only optimizations that do not require any speed-space tradeoffs are used, so the executable should not increase in size. The compiler will take longer to compile programs and require more memory than with -O1. This option is generally the best choice for deployment of a program, because it provides maximum optimization without increasing the executable size. It is the default optimization level for releases of GNU packages.

-O3

This option turns on more expensive optimizations, such as function inlining, in addition to all the optimizations of the lower levels -O2 and -O1. The -O3 optimization level may increase the speed of the resulting executable, but can also increase its size. Under some circumstances where these optimizations are not favorable, this option might actually make a program slower.

-funroll-loops

This option turns on loop-unrolling, and is independent of the other optimization options. It will increase the size of an executable. Whether or not this option produces a beneficial result has to be examined on a case-by-case basis.

2. Know your compiler

GNU compiler important optimization options (2/2):

-Os

This option selects optimizations which reduce the size of an executable. The aim of this option is to produce the smallest possible executable, for systems constrained by memory or disk space. In some cases a smaller executable will also run faster, due to better cache usage.

-march= (pentium4|nocona|athlon-mp|opteron|…)

[NB: athlon-mp is for operon if GCC < 3.4]

The default architecture is i386. GCC runs on all other i386/x86 architectures, but it can result in degraded performance on more recent processors. If you're concerned about portability of an image, you should compile it with the default. If you're more interested in performance, pick the architecture that matches your own.

-mfpmath= (387|sse|sse2)

The default choice is -mfpmath=387 (Standard 387 Floating Point Coprocessor). An experimental option is to specify both sse and 387 (-mfpmath=sse,387), which attempts to use both units.

-ffast-math

Optimization that provides transformations likely to result in correct code but it may not adhere strictly to the IEEE standard. Use it, but test carefully.

Further reading: http://www.linuxjournal.com/article/7269

2. Know your compiler

INTEL compiler important optimization options (1/2):

-O0

This option disables all types of optimizations. It is recommended to use this in early stages of development, until we know that our application is working correctly.

-O1 or -O

This option optimizes for speed bearing in mind the size of code. Suitable for very large code size where in the focus is not on performing iterations (loops). The high level optimizations it performs are as follows: disables software pipelining, disables loop unrolling,global code scheduling, enables optimization for server applications (straight line and branch like with not too many branches).

-O2 (default)

Option '–O2' (alphabet capital 'O' & number two): This is the default level of optimization (and also the recommended level in most case). It creates the fastest code in most cases but could increase the executable code size. It is suitable for typical integer applications that do not use a lot of floating point math. The high level optimizations it performs are as follows: Global code scheduling, Software pipelining, Predication, Control Speculation, Dead code elimination and Dead-store elimination, Loop unrolling, Partial Redundancy elimination, Exception handling optimizations, Structure alignment lowering and optimizations.

-O3

Enables all "-O2" optimization as well as more optimizations suitable for loop intensive code (a lot of iterations) that does a lot of floating point arithmetic on large data sets. Better performance that the "-O2" option is not guaranteed unless there are a lot of iterations in the code and large data sets are involved with a lot of floating point arithmetic. The high level optimizations it performs are as follows: All the optimization done by the "-O2" option, Data pre-fetching, Loop and memory access transformation, Scalar replacement.

-fast

The -fast option maximizes speed across the entire program.For systems based on Xeon processors, including those with EM64T, -fast is equivalent to -O3, -ipo, -static, and -xP

2. Know your compiler

INTEL compiler important optimization options (2/2):

-ipo

Interprocedural optimizations, including selective inlining, among multiple source files.

-ax{K|W|N|B|P}

Automatic Processor Dispatch. Generates specialized code and enables vectorization for the indicated processors while also generating non-processor-specific code. You can use more than one letter to tune for multiple processors in the same executable.

W – Intel Pentium 4 processors and AMD Athlon 64 and Opteron* processors, and processors supporting Intel® EM64T.

P – Code is optimized for Intel® Core™ Duo processors, Intel® Core™ Solo processors, Intel® Pentium® 4 processors with Streaming SIMD Extensions 3 (SSE3), compatible Intel processors with SSE3, and processors supporting Intel® EM64T. The resulting code may contain unconditional use of features that are not supported on other processors.

N – Code is optimized for Intel Pentium 4 and compatible Intel processors with Streaming SIMD Extensions 2 (SSE2). The resulting code may contain unconditional use of features that are not supported on other processors.

-x{K|W|N|B|P}

Processor-specific targeting. Generates specialized code for the indicated processor and enables vectorization. The executable should only be run on the targeted compatible processors.

-mtune=[proc]

Targets optimization for specified processor, but produces code that will run on any processor. Possible values of [proc] include pentium4 (default), pentium, pentiumpro, and pentium-mmx.

2. Know your compiler

PGI compiler important optimization options:

-O0 –O1 –O2 (default) –O3
You should by now know what they are.

-fast
Chooses generally optimal flags for the target platform.

-fastsse
Chooses generally optimal flags for a processor that supports the SSE (Pentium 3/4, AthlonXP/MP, Opteron) and SSE2 (Pentium 4, Opteron) instructions.

-tp k8-64
Target optimization to the 64-bit Opteron processor.

-mp
Enable the compiler to generate multi-threaded code based on the OpenMP directives.

Performance issues:
how to get the best from your x86, x86_64 or em64t system?

1. Know your processor
2. Know your compiler
3. **Know your application**

3. Know your application

Power is nothing without control.

Although compilers generally produce very compact object code, many performance improvements are possible by careful source code optimization. Most such optimizations result from taking advantage of the underlying mechanisms used by compilers to translate source code into sequences of instructions.

This argument could be covered in a semester course, or more. I will show only a couple of examples.

3. Know your application

Explicit Parallelism in Code

Optimization
Where possible, break long dependency chains into several independent dependency chains that can then be executed in parallel, exploiting the execution units in each pipeline.
Rationale and Examples
This is especially important to break long dependency chains into smaller executing units in floating-point code, because of the longer latency of floating-point operations. Because most languages (including ANSI C) guarantee that floating-point expressions are not reordered, compilers cannot usually perform such optimizations unless they offer a switch to allow noncompliant reordering of floating-point expressions according to algebraic rules.

Avoid

```
double a[100], sum;int i;
sum = 0.0f;
for (i = 0; i < 100; i++) {
sum += a[i]; }
```

Preferred

```
double a[100], sum1, sum2, sum3, sum4, sum;
int i;
sum1 = 0.0;
sum2 = 0.0;
sum3 = 0.0;
sum4 = 0.0;
for (i = 0; i < 100; i + 4) {
sum1 += a[i];
sum2 += a[i+1];
sum3 += a[i+2];
sum4 += a[i+3];}
sum = (sum4 + sum3) + (sum1 + sum2);
```

Notice that the four-way unrolling is chosen to exploit the four-stage fully pipelined floating-point adder. Each stage of the floating-point adder is occupied on every clock cycle, ensuring maximum sustained utilization.

3. Know your application

Replacing Integer Division with Multiplication

Optimization
Replace integer division with multiplication when there are multiple divisions in an expression. (This is possible only if no overflow will occur during the computation of the product. The possibility of an overflow can be determined by considering the possible ranges of the divisors.)

Rationale
Integer division is the slowest of all integer arithmetic operations.

Examples

Avoid code that uses two integer divisions:

```
int i, j, k, m;
m = i / j / k;
```

Instead, replace one of the integer divisions with the appropriate multiplication:

```
m = i / (j * k);
```

References

- Scott, L. Ridgway "Bioinformatics“. Perspectives in Biology and Medicine, Volume 47(1),135-139, 2004.
- Luciano da Fontoura Costa, Bioinformatics: perspectives for the future. Genet. Mol. Res. 3 (4): 564-574 (2004)
- C. Arlandini, GOLGI: un cluster Opteron per il CILEA. Bollettino del CILEA April (101),9-12, 2006.
- Richard S. Morrison, Cluster Computing Architectures, Operating Systems, Parallel Processing & Programming Languages, 2003.
- AAVV, Software optimization guide for AMD64 processors, AMD Publication #25112, Rev. #3.06, September 2005