High Performance Computing

ELIXIR-EXCELERATE Train-the-Researcher HPC course

Oswaldo Trelles

ortrelles@uma.es

excelerate

European Life Sciences Infrastructure for Biological Information

www.elixir-europe.org

ELIXIR-EXCELERATE is funded by the European Commission within the Research Infrastructures programme of Horizon 2020, grant agreement number 676559.

Contents

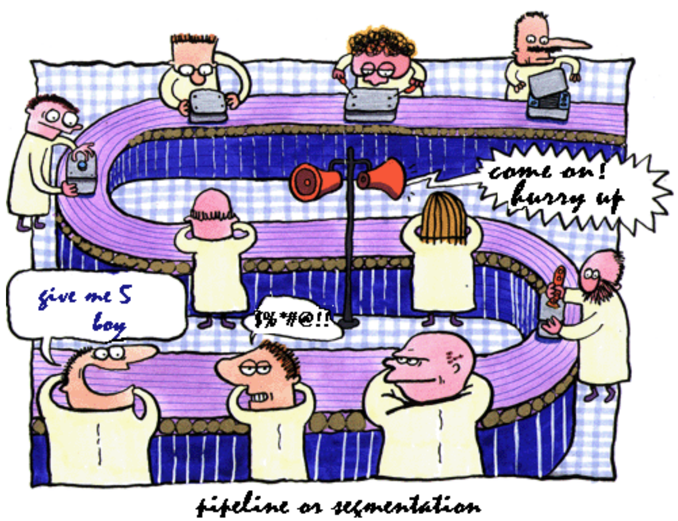
Introduction to HPC concepts (theoretical part)

- Parallel architectures
- Parallel programming languages and paradigms
- Program analysis, program transformations, data locality optimizations
- Parallelization techniques, run-time systems, software environments
- Master-slave model
- Map – Reduce techniques
- Load distribution and balancing
- Scheduling by priorities

HPC ≈ Parallel computing

"Parallel computing consists in the use of a collection of processing elements that cooperate in a concurrent way with the aim of offering a better behavior than conventional (sequential) computing in some aspect"

"A *parallel computer* is a set of processors that are able to cooperate in solving computational problems"

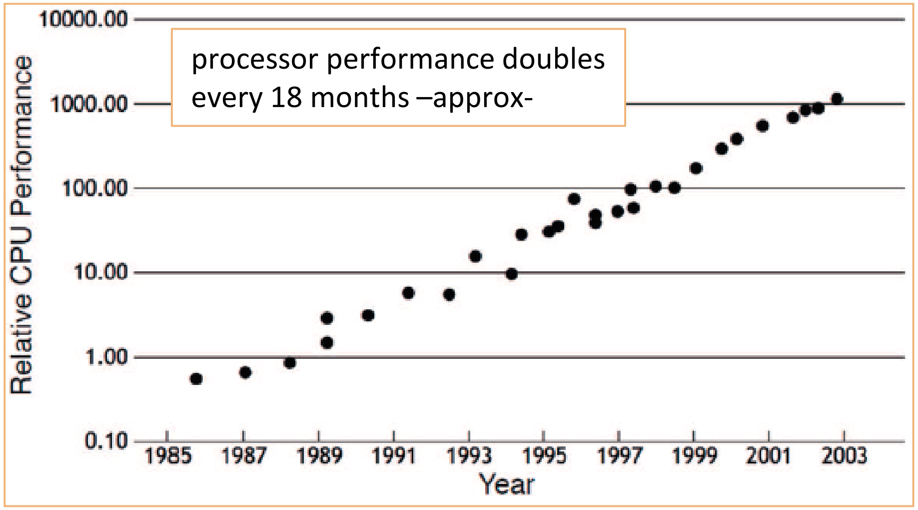


Why parallel computing ?

To get performance improvements that exceeds what processor provide

- Increasing clock frequency
- Exploiting instruction-level parallelism: Pipelining, ILP (superscalar)

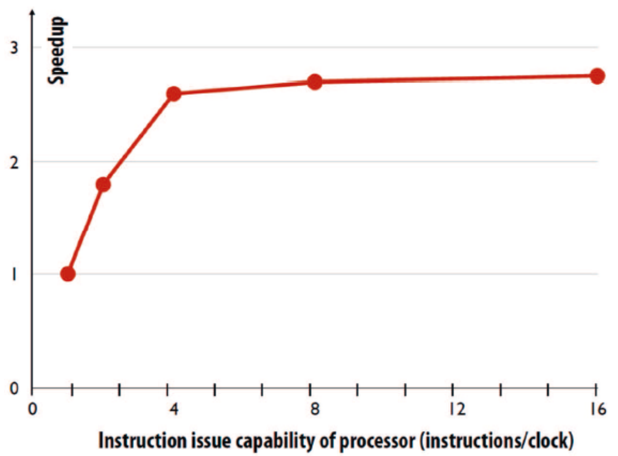
- More computing capacity to address new problems (not only larger, but more complex)
- Explore exhaustive algorithms instead of heuristics (solve problems with finer accuracy)
- Reduce computing response time
- Provide concurrency
- Strategic tool for innovation



End of ILP and Frequency Scaling

Single-instruction stream performance scaling has decreased

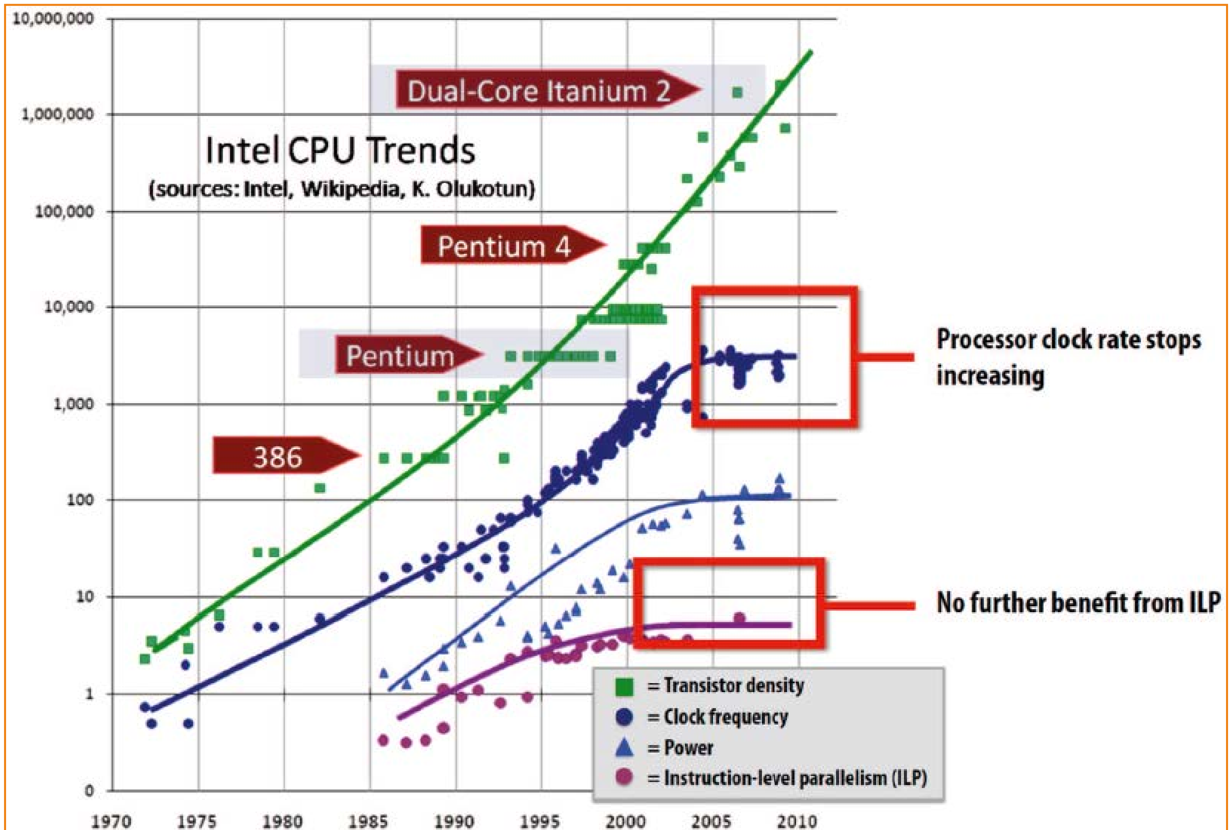
- **ILP wall**: No further improvements by more ILP (Data dependencies, branch prediction, speculation …)
- **Power wall**: The increase in energy consumption is unsustainable (Power density is so high to be effectively dissipated
- **Memory wall**: The processor-memory latency gap continue to increase



End of ILP and Frequency Scaling

Most available ILP is exploited by a processor capable of issuing only **four** instructions per clock cycle

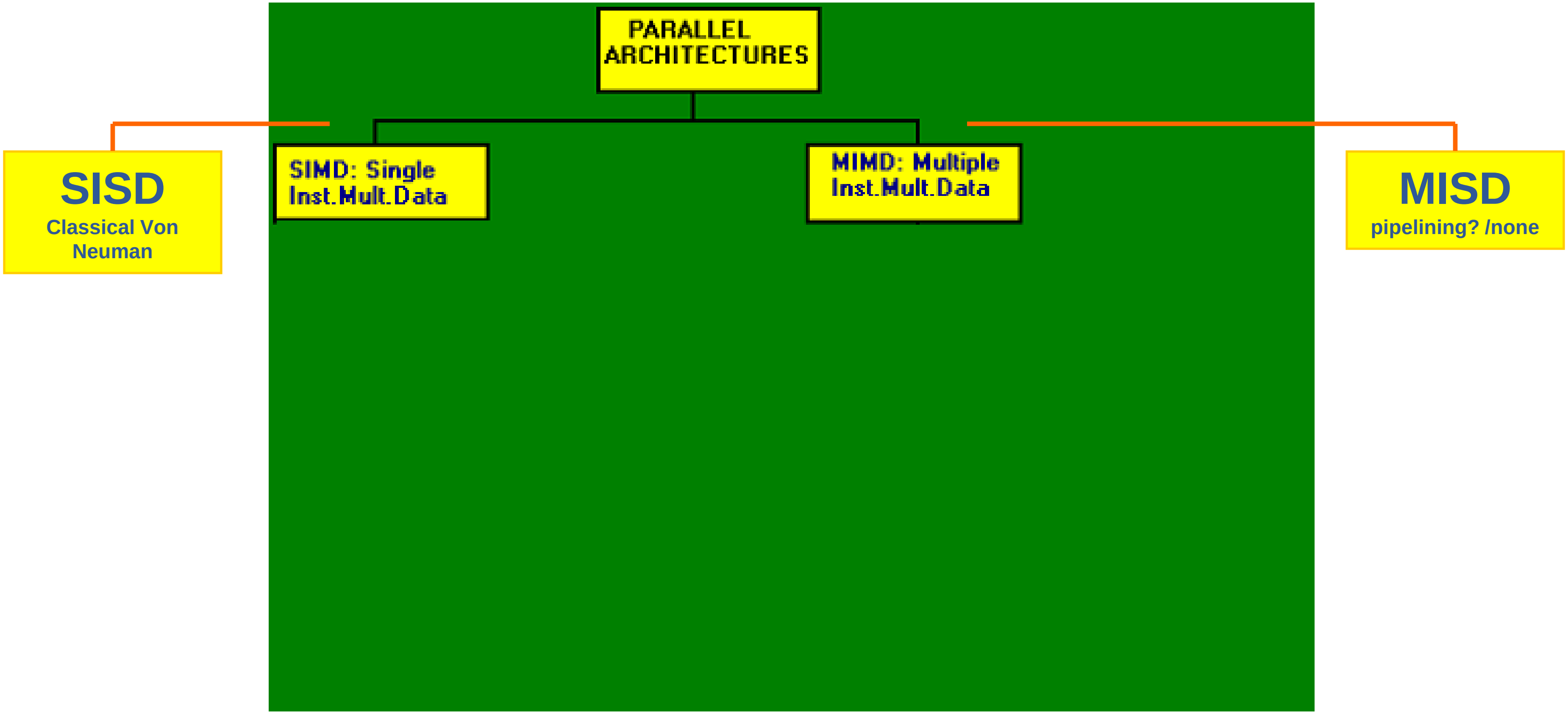
Fetch – Decode – Operands – Execute - Store



Multicore performance scaling

- From single-core to multi-core processors
- Architects are now building faster processors by adding more execution units that run in parallel
- Software must be written to be parallel to see performance gains
- Task-level parallelism (multi-threading)
- Processors include multiple execution cores in a single chip

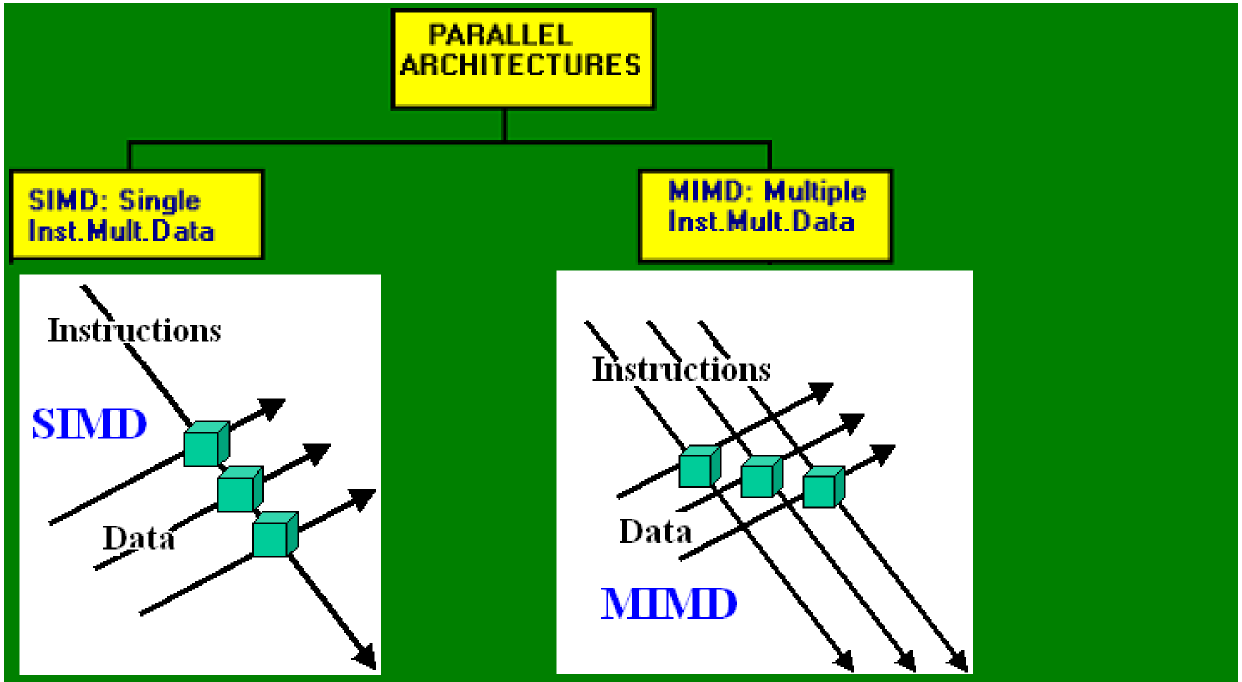
Architecture taxonomy



Flynn’s Classification

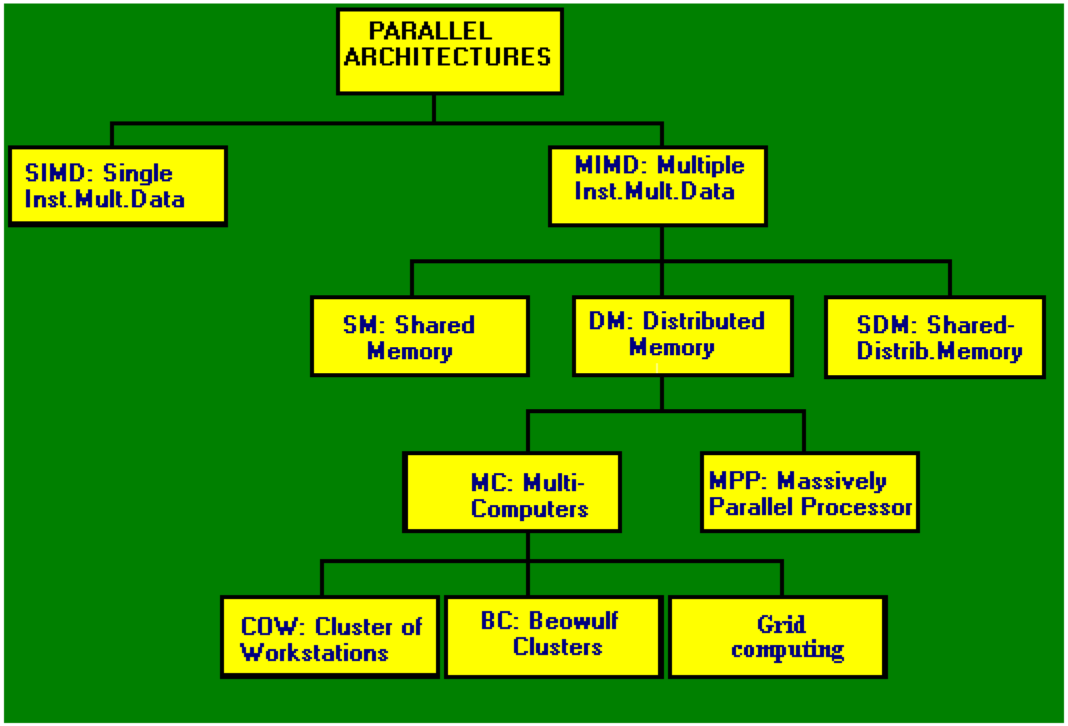
(Instruction & data stream

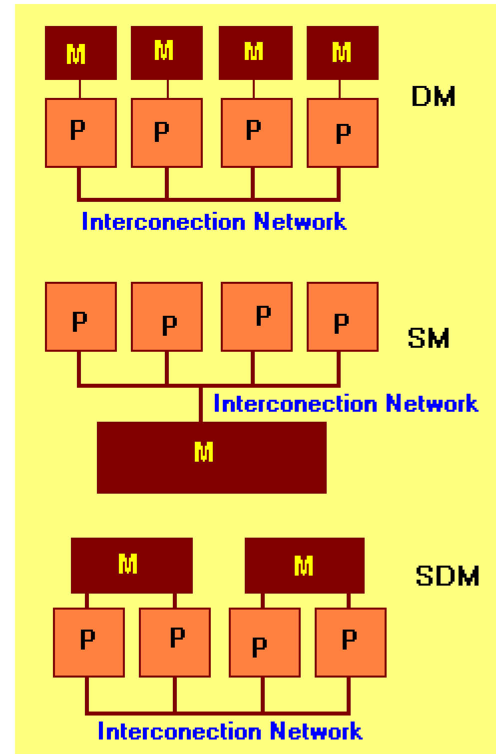
Architecture taxonomy



SIMD machines	MIMD machines
Many simple processors/small local memory.	Limited number of PEs (scalability)
The same instruction over different data	Each PE executes *asynchronously*
Synchronized and explicit communications.	Independence between processes
Programming complexity and often inflexibility	Strong influence of Memory Architect.
Strong dependency on synchronization	Memory is a key issue
Restricted to special-purpose applications.	More amenable to bioinformatics

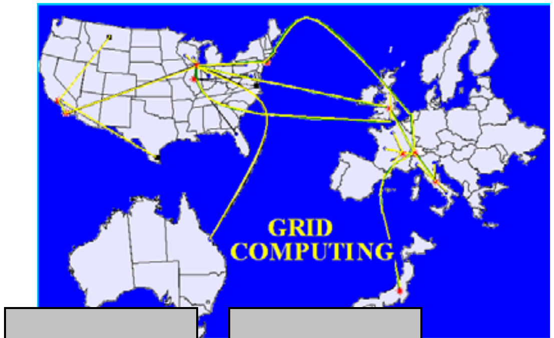
Architecture taxonomy

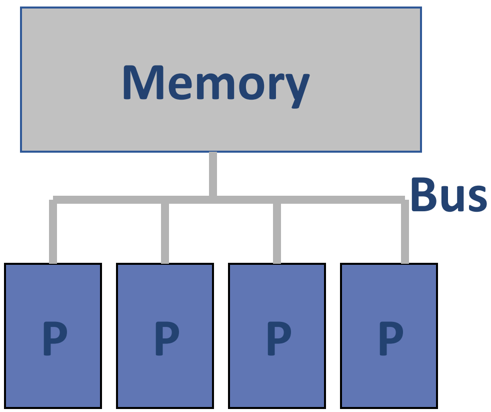


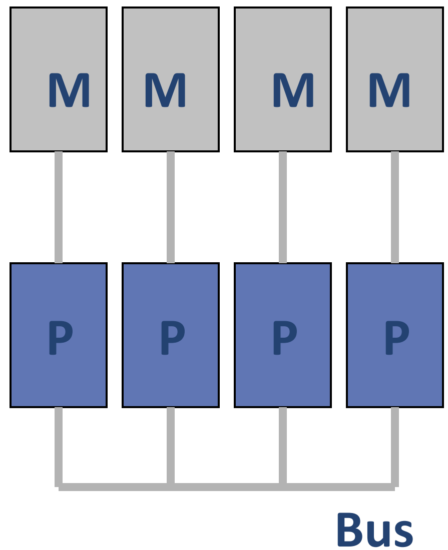


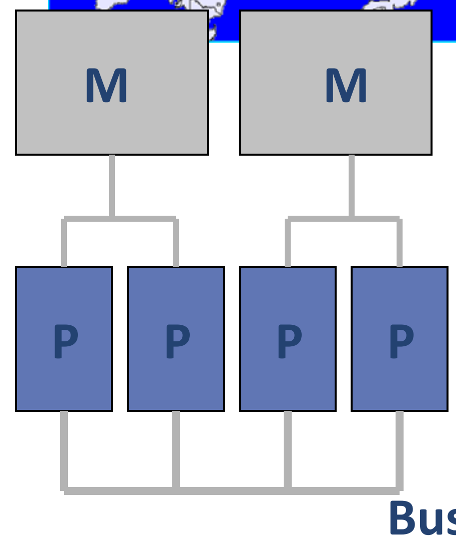
Multiprocessor Architectures

Cluster of WorkStations







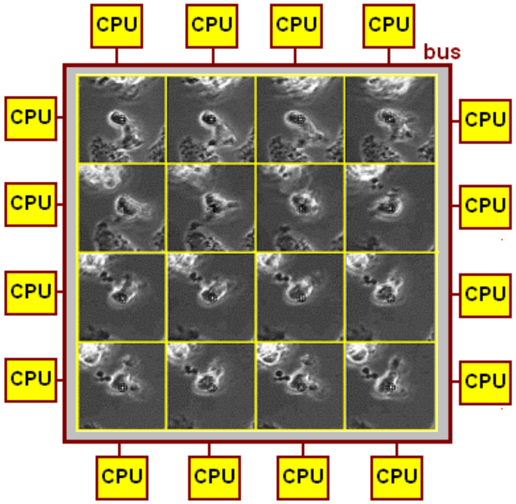


Shared Memory

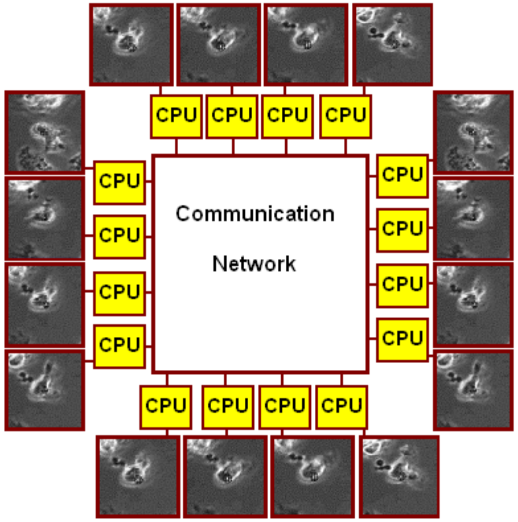
Distributed Memory

Distributed-Shared Memory

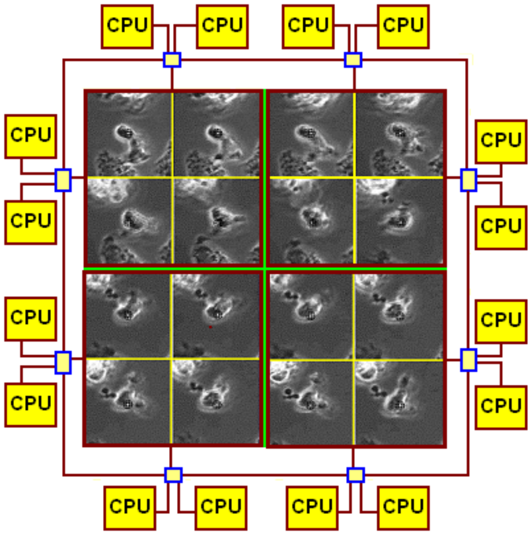
Multiprocessor Architectures



Shared Memory



Distributed Memory

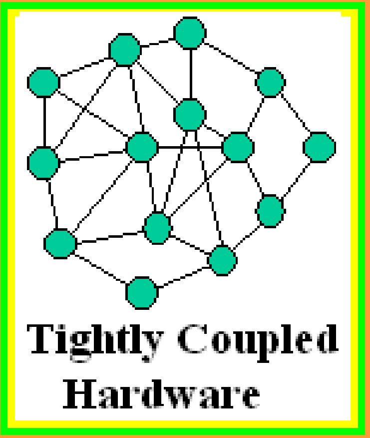


Distributed-Shared Memory

MIMD : Memory Architecture

Shared-memory architecture

Any process, in any processor, has direct access
to any local or remote memory in the system



- Single address map (simple programming).
- No 'time penalty' for communication (UMA)
- Scalability drawbacks.

Distributed memory systems

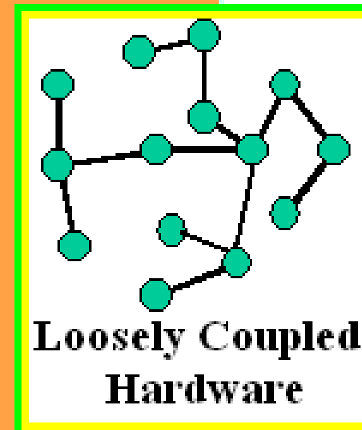
- Scalability
- Communication penalty (NUMA architecture).

Shared-Distributed Memory

- The best of both memory architectures
- Short memory in each node (DM) + hardware –routers- support (SM).
- Slight time penalty.

MIMD : Memory Architecture

Parallel Virtual Machines: a dynamic network of computing resources that work together as a single, uniform operating environment.



Multi-Computers : Fast microcomputers connected by a LAN

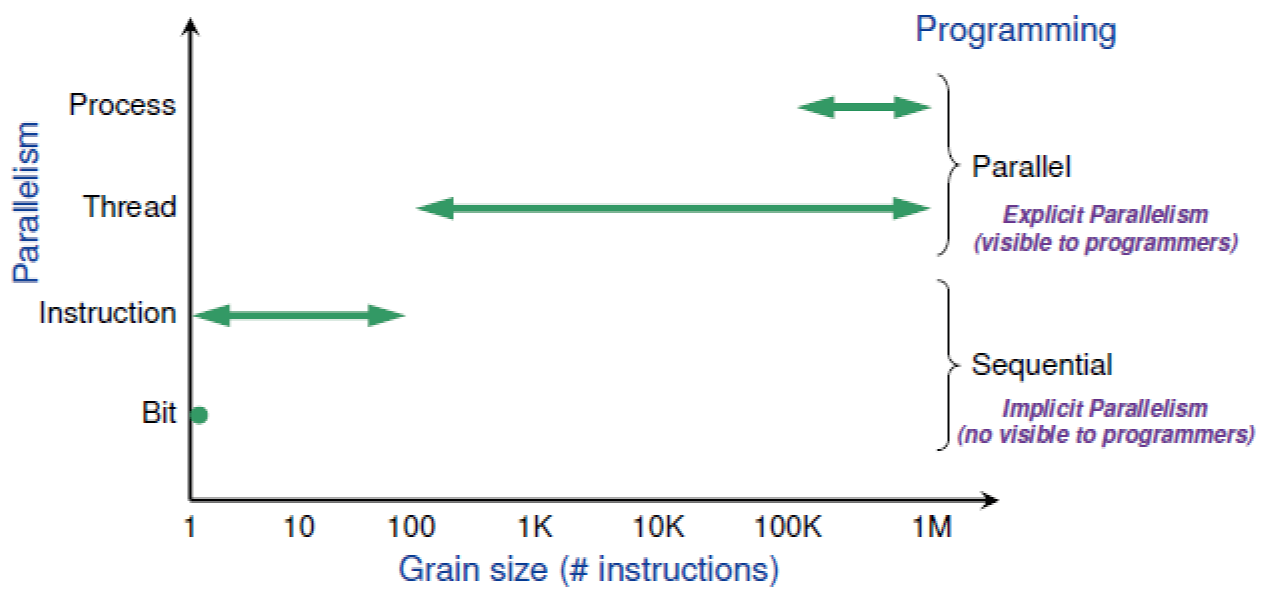
- Distributed (loosely coupled) environment.
- Communication: a key issue
- Scalability
- Low cost platforms: COWs, beowulf-clusters: PCs+ pd soft (Linux, PVM,…)

Grid computing: Geographically distributed Services:

- resource discovery / resource scheduling / uniform computing & data access
- authentication, delegation, and secure communication (Grid security services)
- system management and access

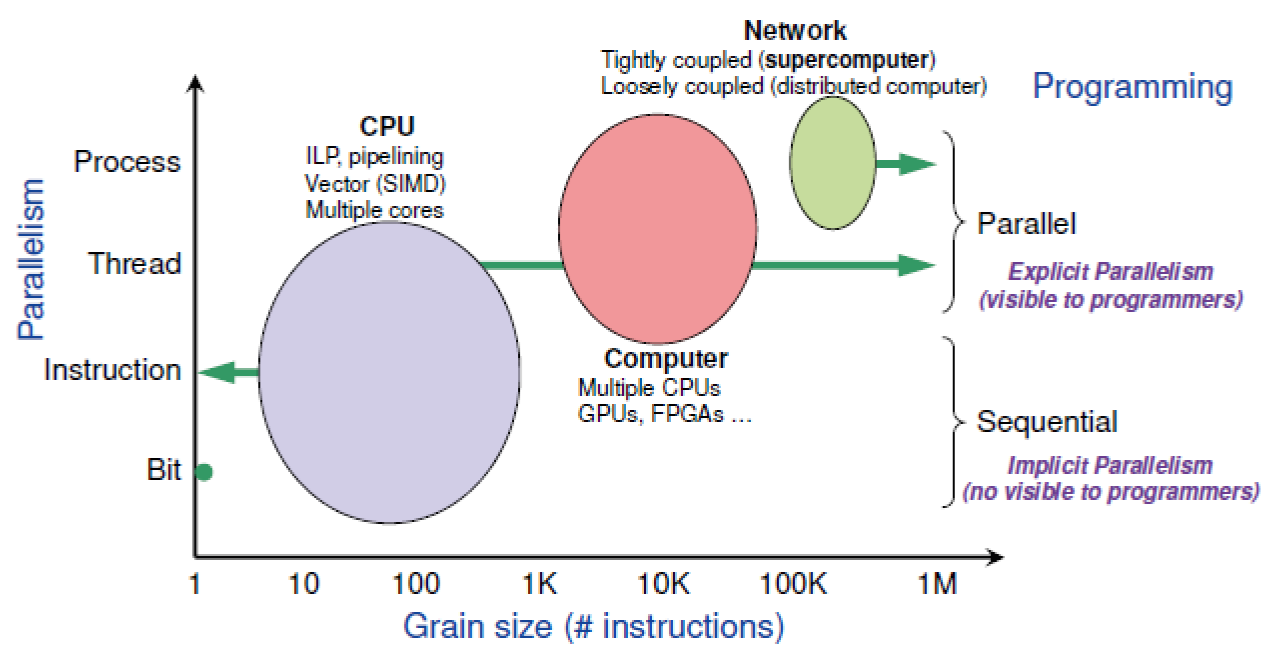
Parallel Platform

- General Parallel Strategy
- From regular to irregular algorithms
- Portability
- Shared, distributed and S/D memory architectures
- Implementation cost
- Task Parallel model



Parallel Platform

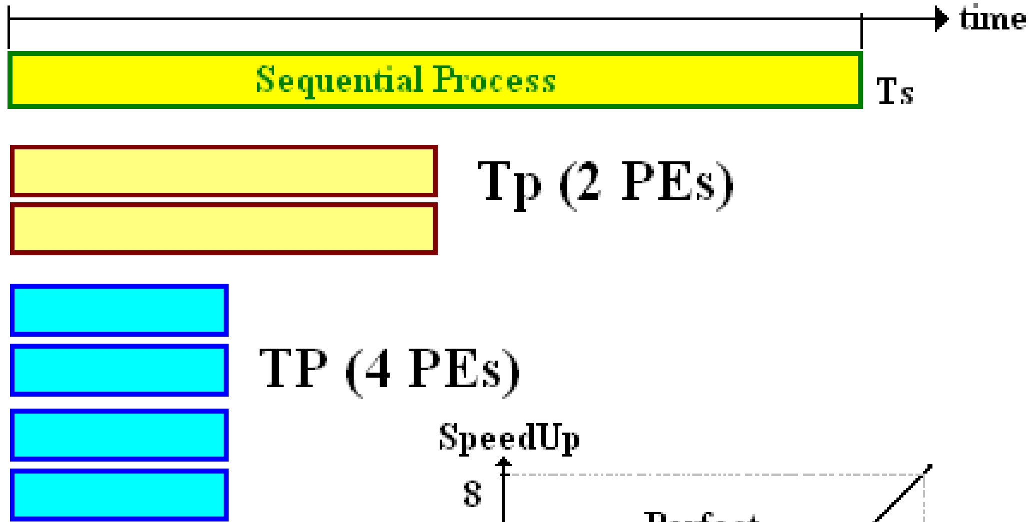
- General Parallel Strategy
- From regular to irregular algorithms
- Portability
- Shared, distributed and S/D memory architectures
- Implementation cost
- Task Parallel model



Parallel Programming Concepts

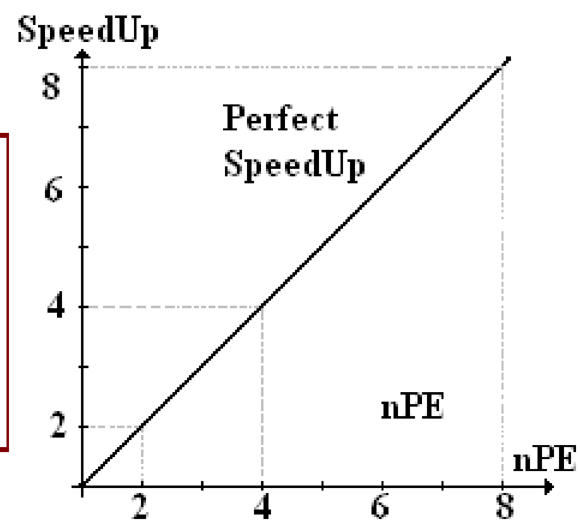
- **Granularity**: the relative size of units of computation (Coarse/fine)
- **Communication**: Data Exchange and synchronization
- Shared Memory (critical sections)
- Message-Passing (where the data are, what to communicate, when to whom)
- **Task Scheduling** (master/slave models, pipeline)

Parallel Programming Concepts: Speed-Up



$$SpeedUp = \frac{Ts}{Tp}$$

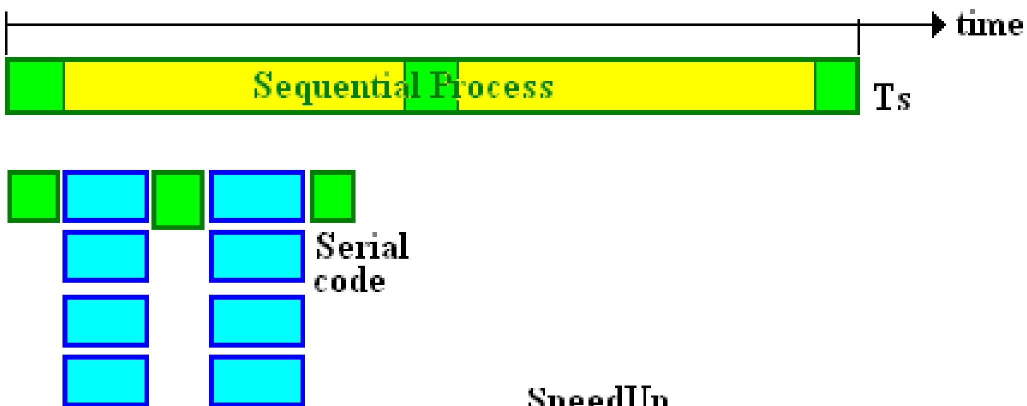
$$Tp(ideal) = \frac{Ts}{nPE}$$



Ideally, if we have *n* processors, the run time should also be *n* times faster

Basic parallel concepts: Amdahl's Law

the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.



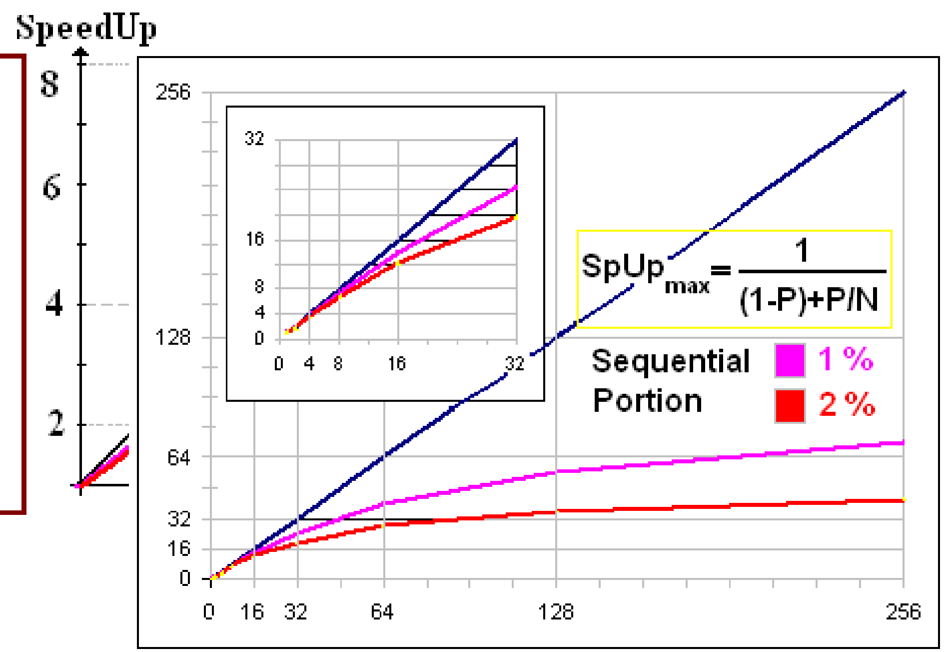
Amdahl's Law

(no load balance problems)

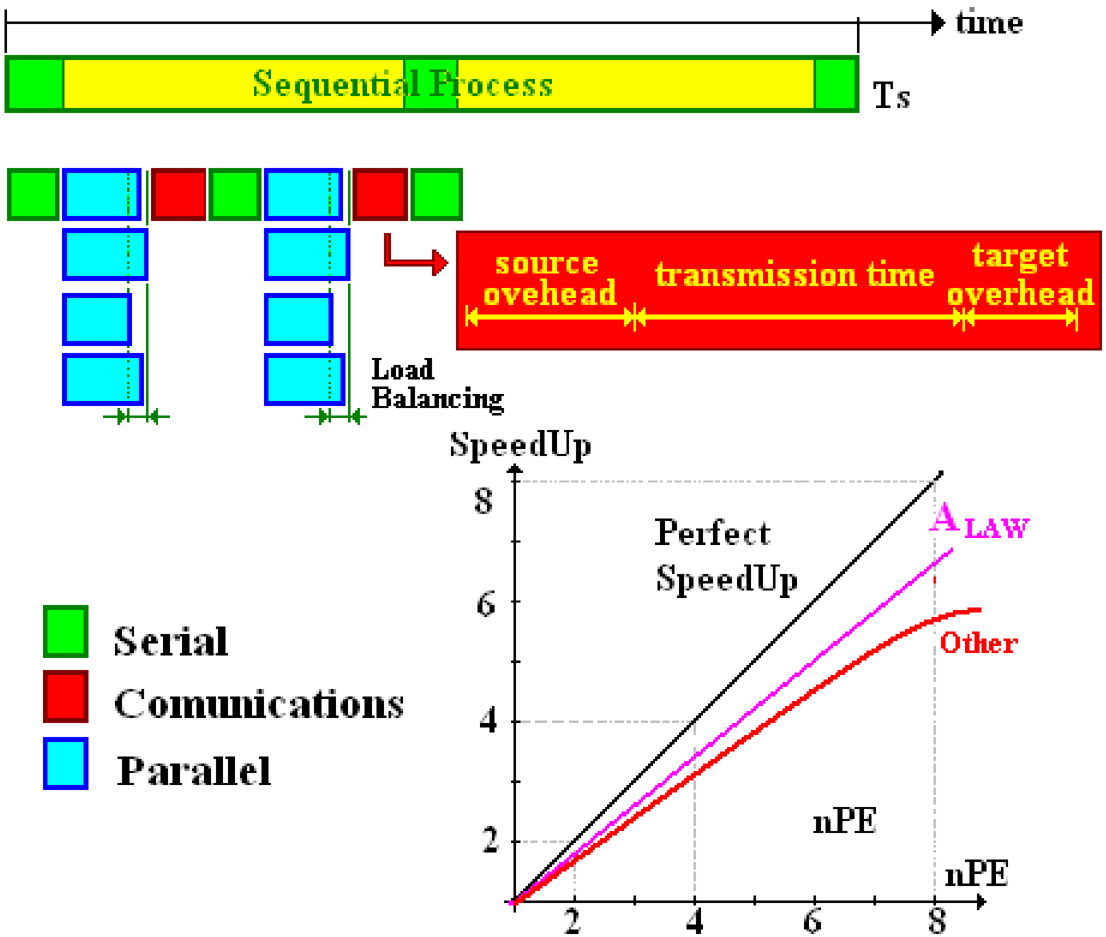
$$SpeedUp_{max} = \frac{1}{(1-P)+P/N}$$

P = Parallel portion of code

N= Number of processors



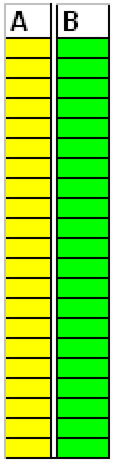
Basic parallel concepts: Sources of Inefficiency



Performance decreases by:

- Serial calculations
- Synchronization
- Interaction (communication)
- Load imbalance (idle PEs)
- Task Scheduling

Basic parallel concepts: an example



C

$$C = \sum_{1}^{n} A(i) * B(i)$$

```
//   dot (scalar) product:
     c=0.0
     do i=1,1000
            c=c+a(i)*b(i)
     end do
     c=sqrt(c)
```

```
// Parallel Version (2 PEs):
If (pID == 0) c=0.0
// synchronization
do i=id*500+1,(id+1)*500
     c=c+a(i)*b(i)
enddo
// synchronization
If (pID == 0) c=sqrt(c)
```

Take care!

Basic parallel concepts: an example

```
// Load Distribution:
Load=(1000+ n - 1) / n
From = pID * Load+1
To    = min( (pID+1)*Load, 1000)
do i= From, To
```

```
// Parallel Version (n PEs):
!$omp single
c=0.0
!$omp end single
!$omp pdo private(cl)
do i=From,To
    cl=cl+a(i)*b(i)
enddo
!$omp end pdo
!$omp atomic
c=c+cl
!$omp barrier
if(id==0)c=sqrt(c)
```

Basic parallel concepts: an example

```
// Load Distribution:

Load=(1000+ n - 1) / n

From = pID * Load+1

To    = min( (pID+1)*Load, 1000)

do i= From, To
```

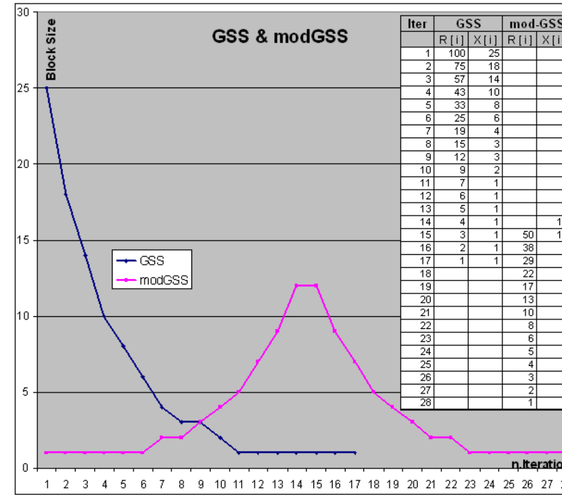
```
// Parallel Version
If (pID==0) bradcast(A,B,n)
          else recv(A,B,n);
c=0.0
do i= From, To
     c=c+a(i)*b(i)
enddo
if (pID!=0) send(pID,c)
 else {
     do j=1,n
       recv(pIDs,d)
       c=c+d
     enddo
     c=sqrt(c)
 }
```

Message-Passing

send and *revc* are blocking primitives

Use cases

Database searching: master - workers



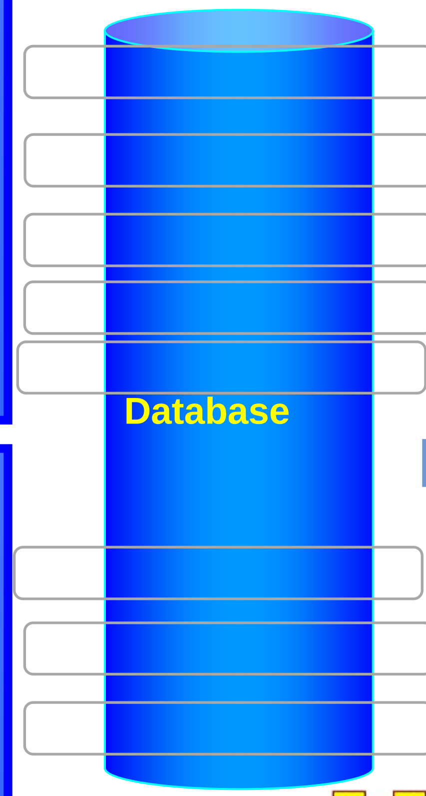
Iter	GSS		mod-GSS	
	R [i]	X [i]	R [i]	X [i]
1	100	25		1
2	75	18		1
3	57	14		1
4	43	10		1
5	33	8		1
6	25	6		1
7	19	4		2
8	15	3		2
9	12	3		3
10	9	2		4
11	7	1		5
12	6	1		7
13	5	1		9
14	4	1		12
15	3	1	50	12
16	2	1	38	9
17	1	1	29	7
18			22	5
19			17	4
20			13	3
21			10	2
22			8	2
23			6	1
24			5	1
25			4	1
26			3	1
27			2	1
28			1	1

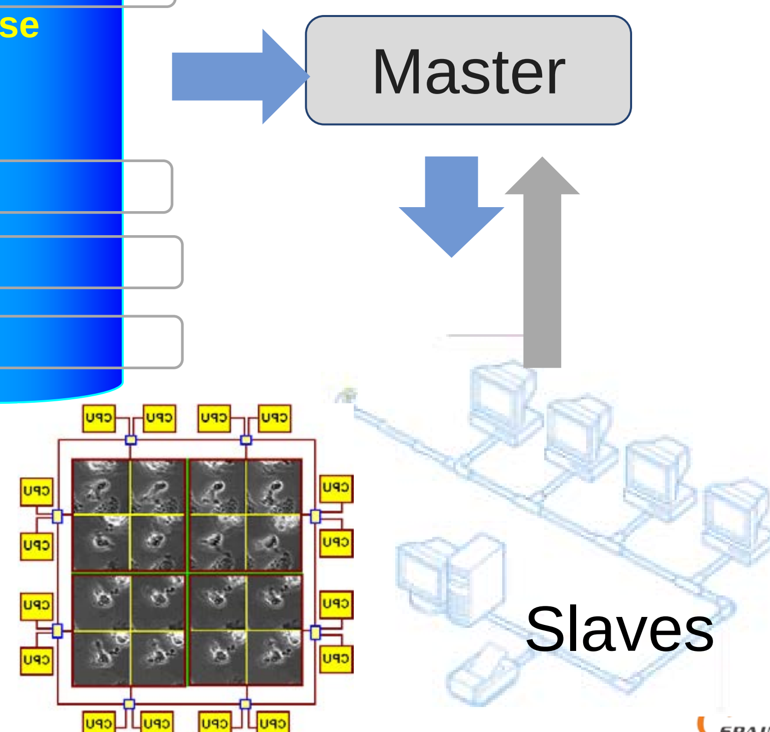
DB-searching Applications

- High number of tasks
- Heterogeneity (tasks & CPU-power)
- Network overload
- Scheduling / distribution overload
- Task synchronization
- Fault tolerance
- Portability

The model

- Task parallel (coarse grained)
- Dynamic load balancing
- Network optimization (message size)
- Minimize number of messages
- Buffering (speculative scheduling)
- Check-points
- SM, DM, D&SM architectures





Database searching: master - workers

Master

```
Get Parameters, Initialize
Start_Workers
Get QuerySeq
Broadcast(QuerySeq)
While (!eof or TransitMess) {
   for all Free_Workers {
        (!eof) Get DBseq
        Prepare(Message)
        Send(Message)
        TransitMess++;
    }
   Receive(R_mess)
   TransitMess--;
}
Broadcast(END_mess)
Report_Best_Results
```

Workers

```
Start with params
Perform Initializations
Receive (Query_seq)
while (! END_mess) {
  Receive(Message)
  Score=Algorithm(QuerySeq,DBseq,par);

  Send(Results)
}
```

MSA – Clustal: priorities

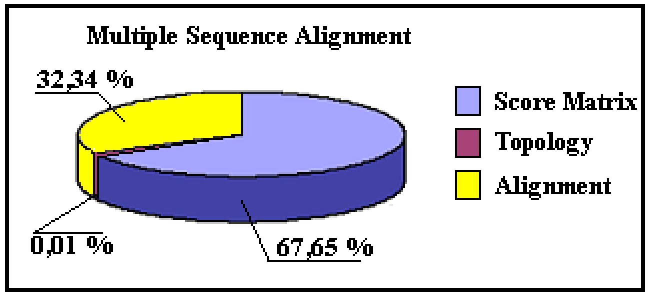
(Thompson J. *et al*, NAR, 1994, 2003, 2007)

Cross similarity matrix (pairwise)

average alignment calculation spends most of its time here
easy to parallelize as all *N*(N-1)/2* elements are independent

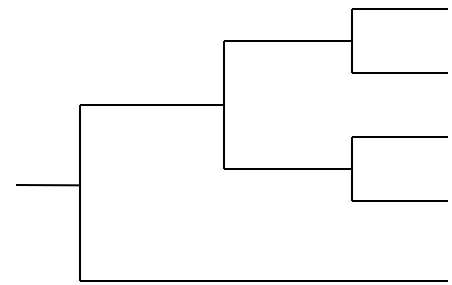
Cross Similarity Matrix

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	-	-	-	-	-	-	-
[1]	82	-	-	-	-	-	-
[2]	52	54	-	-	-	-	-
[3]	60	62	86	-	-	-	-
[4]	22	24	18	24	-	-	-
[5]	26	20	12	16	78	-	-
[6]	22	14	10	8	46	48	-



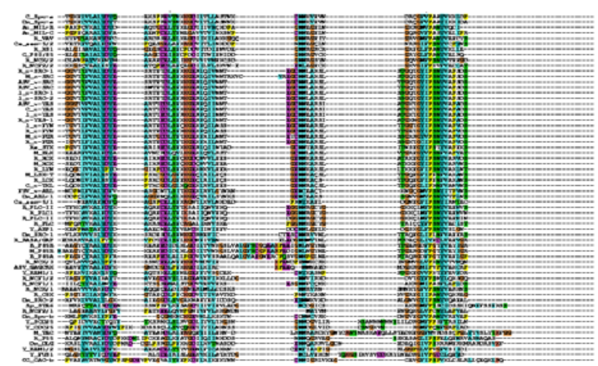
Alignment topology

Calculation of closest sequences (branch) is a relatively light task, that can be solved sequentially.

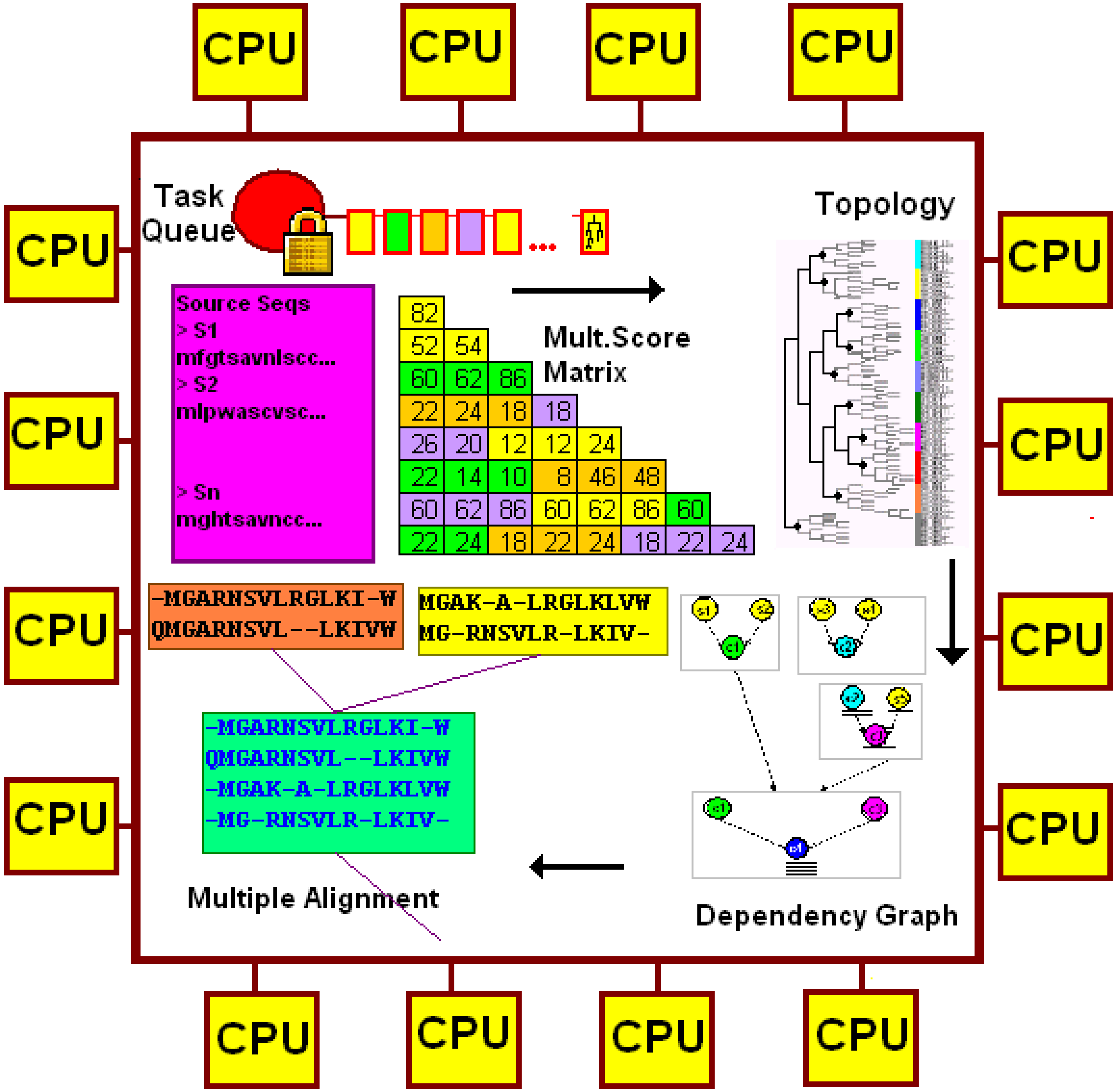


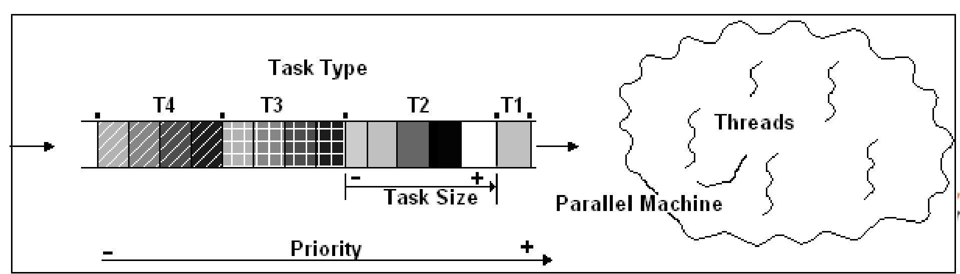
Progressive alignment

Remaining ~30% of the code can be parallelized at this stage by calculating profile scores in parallel, and by solving data dependencies. *(N-1) cluster vs cluster* alignments must be solved.
As a result the whole application is ~90% parallel depending on a size of a problem



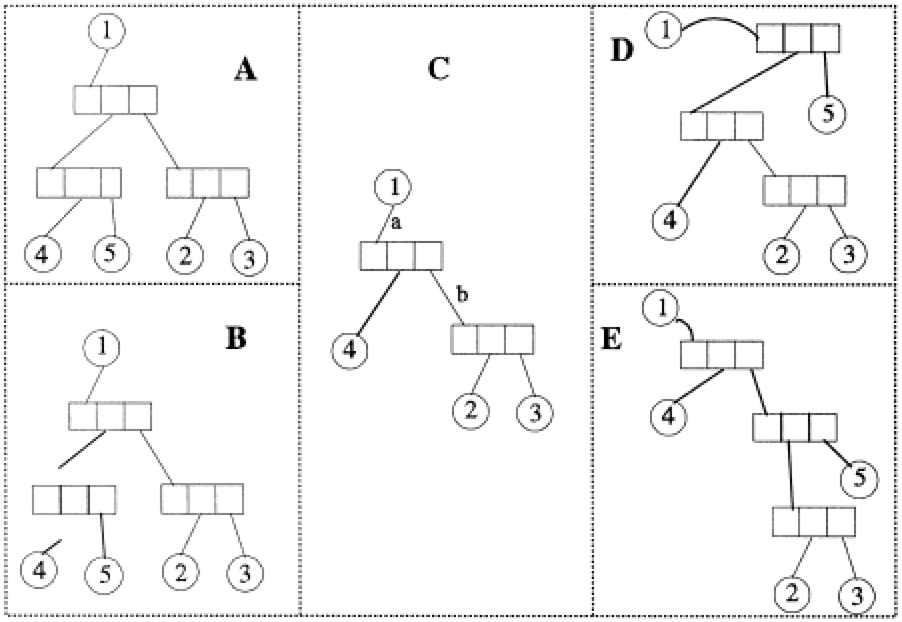
Shared Memory Parallel Model



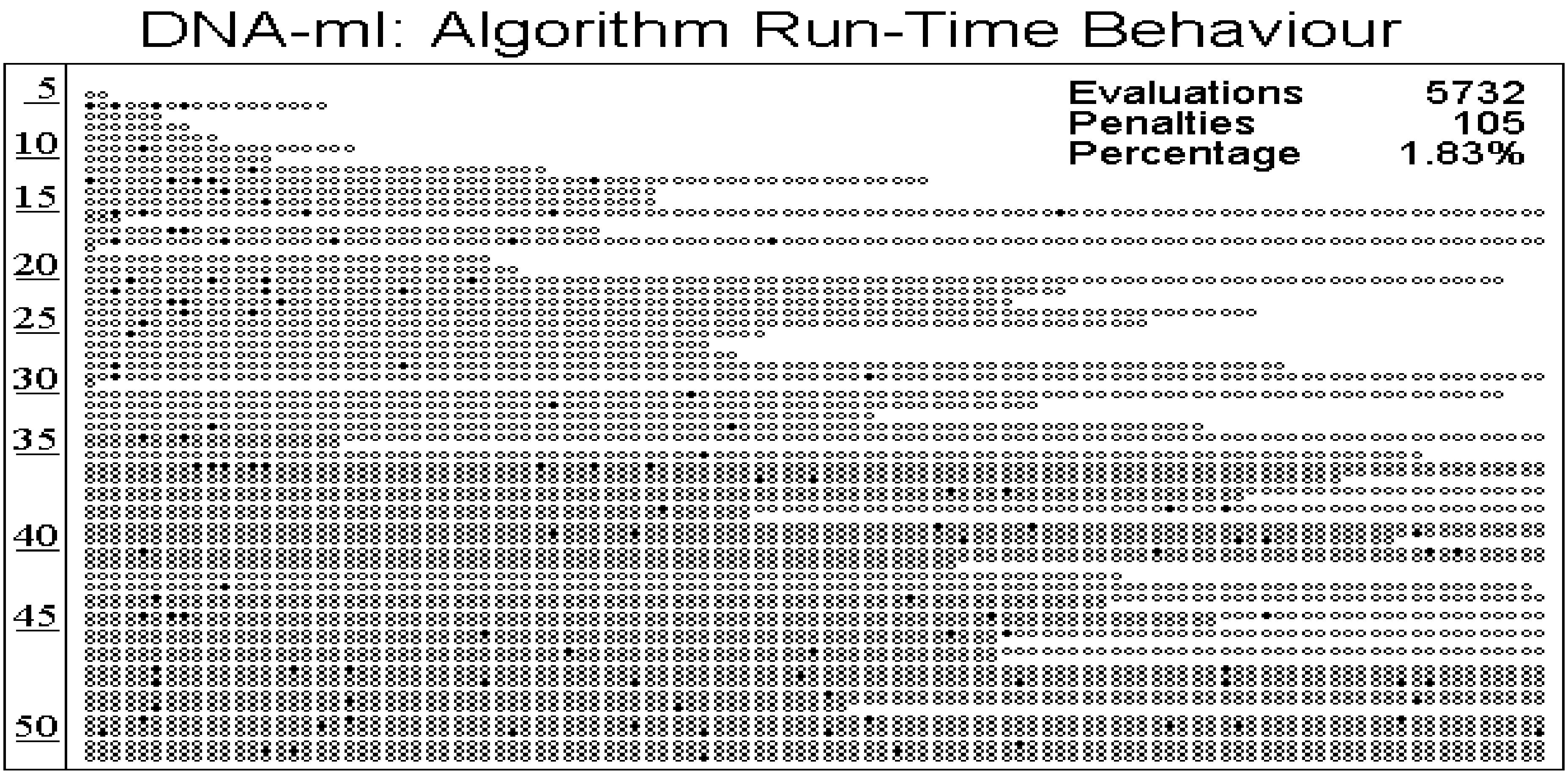


Irregular algorithm: DNAml

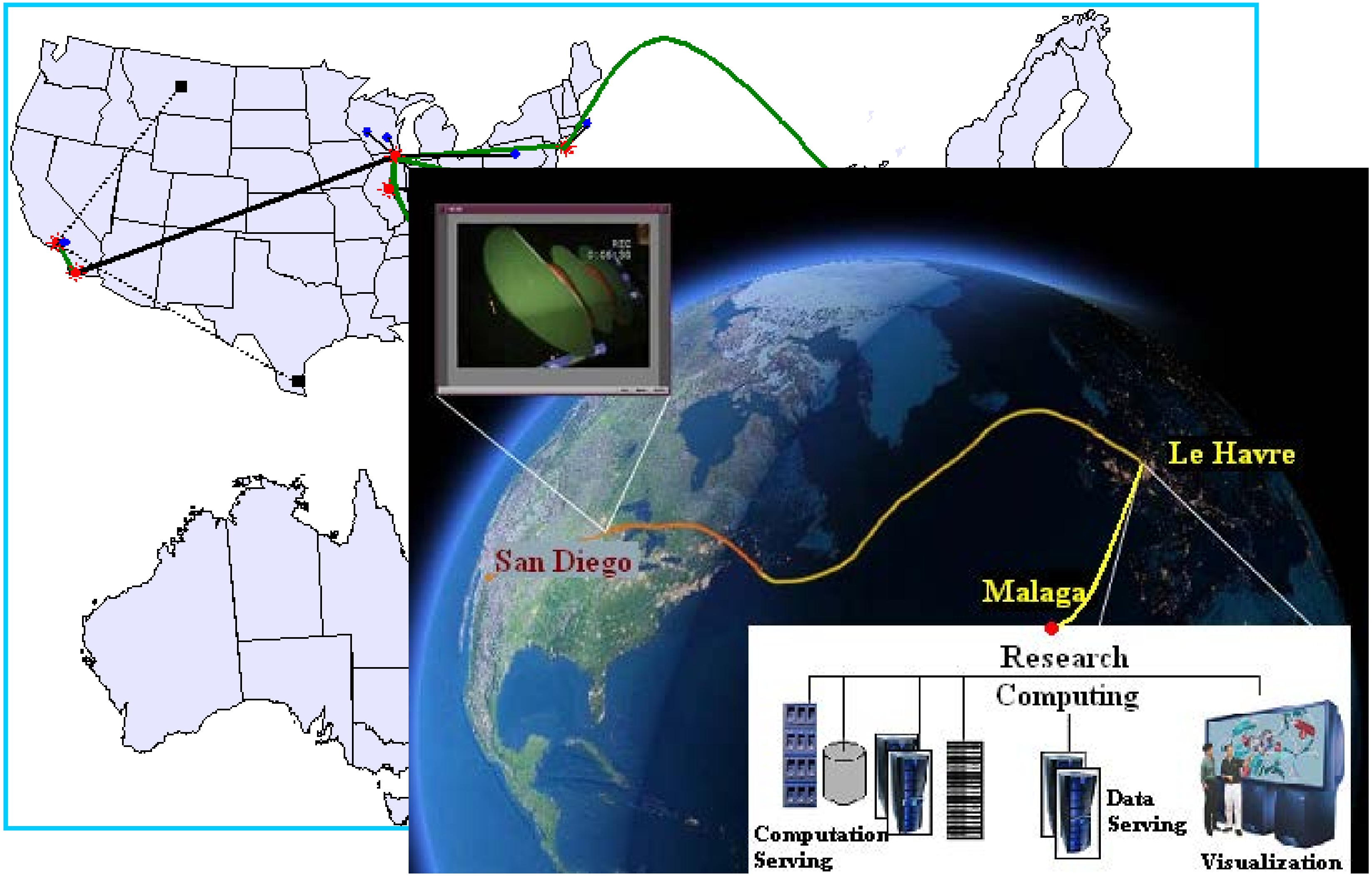
```
A.   Current-best-tree Tk (Lk) [from insertion step]
     for i = 1 to n-tasks
B.     Remove sub-tree i from Tk and produce Tk1 and Tk2
C.     Likelihood evaluation for Tk1 and Tk2 (Lk1 and Lk2)
D-E. Current-best-tree Tk = tree with greater likelihood (Tk, Tk1, Tk2)
      end for
```



Irregular algorithm: Speculative computing

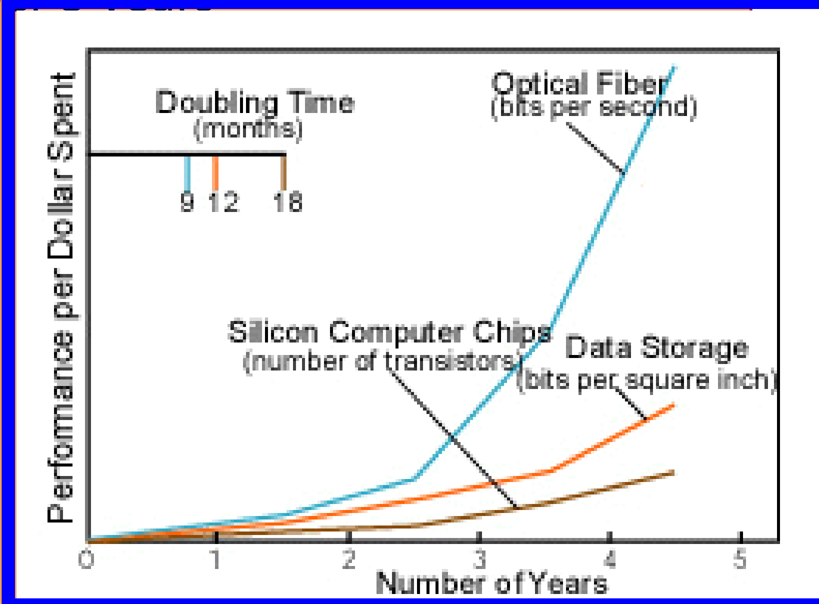


GRID Computing



GRID computing
Network Exponentials

- Network vs. computer performance
  - Computer speed doubles every 18 months
  - Network speed doubles every 9 months
  - Difference = order of magnitude per 5 years
- 1986 to 2000
  - Computers: x 500
  - Networks: x 340,000
- 2001 to 2010
  - Computers: x 60
  - Networks: x 4000



<u>Moore's Law vs. storage improvements vs. optical improvements.</u> Graph from **Scientific American** (Jan-2001) by Cleo Vilett, source Vined Khoslan, Kleiner, Caufield and Perkins.°