Scientific Computing on Parallel Machines

Cálculo científico en ordenadores paralelos

Markus Uhlmann

CIEMAT

www.ciemat.es/sweb/comfos/personal/uhlmann

UCM – Abril 2008

Schedule

Part I	Introduction to parallel programming	lecture 1
Part II	Introduction to MPI	
	General introduction & MPI program structure	lecture 2
	Point-to-point communication	lecture 3,4
	Collective communication	lecture 5
	2D Poisson example	lecture 6,7
	Non-contiguous data & Mixed datatypes	lecture 7
	Virtual topologies & Communication subsets	lecture 8
	Use of linear algebra libraries	lecture 9
	Extensions	lecture 10

Scope of the course

This is a practical course!

- provide the necessary background
- focus on the message passing model
- gain experience by solving examples
- advice on principal considerations

Prerequisites for this course

Parallel programming experience:

- none!

Programming experience:

- C or Fortran
- some knowledge of GNU/Linux or Unix environment useful

Numerics:

- basic knowledge

Schedule

Part I	Introduction to parallel programming	lecture 1
Part II	Introduction to MPI	
	General introduction & MPI program structure	lecture 2
	Point-to-point communication	lecture 3
	Collective communication	lecture 4
	2D Poisson example	lectures 5,6
	Non-contiguous data & Mixed datatypes	lecture 7
	Virtual topologies & Communication subsets	lecture 8
	Use of linear algebra libraries	lecture 9
	Extensions	lecture 10

Small survey

What do you know already about parallel programming?

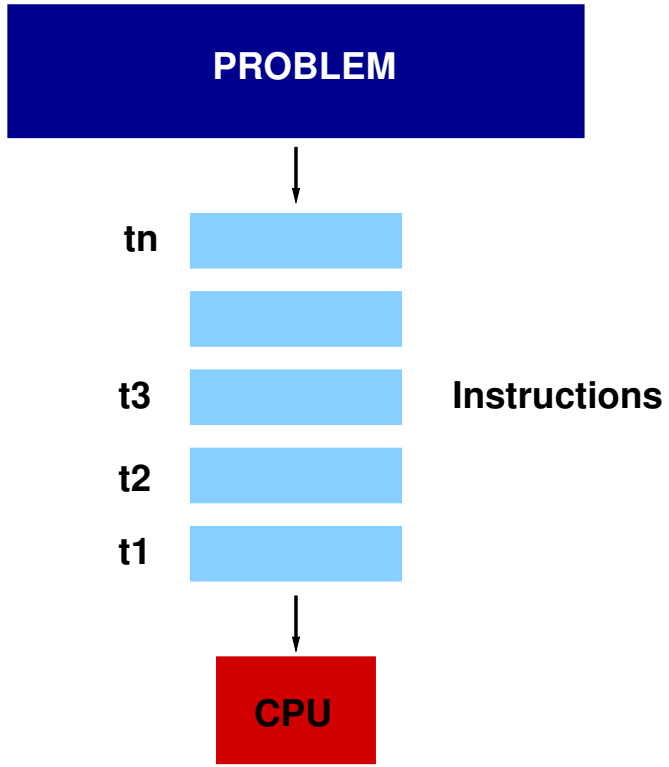
Do you have particular applications in mind?

Part I: Introduction to parallel programming

What is parallel computation?

Serial computing

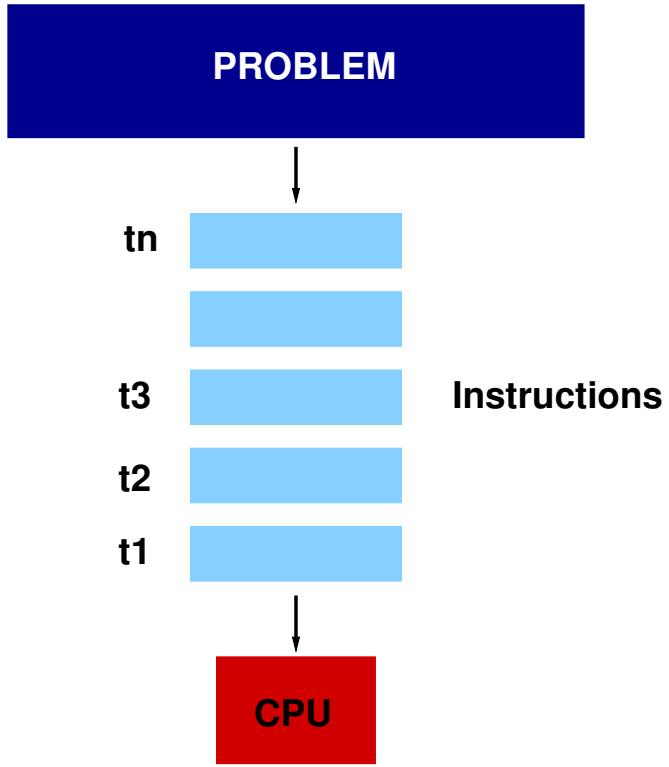
- problem statement
- translate problem into instructions
- use a single processor (CPU)
- execute instructions *serially*: one at a time



What is parallel computation?

Limitations of serial computing

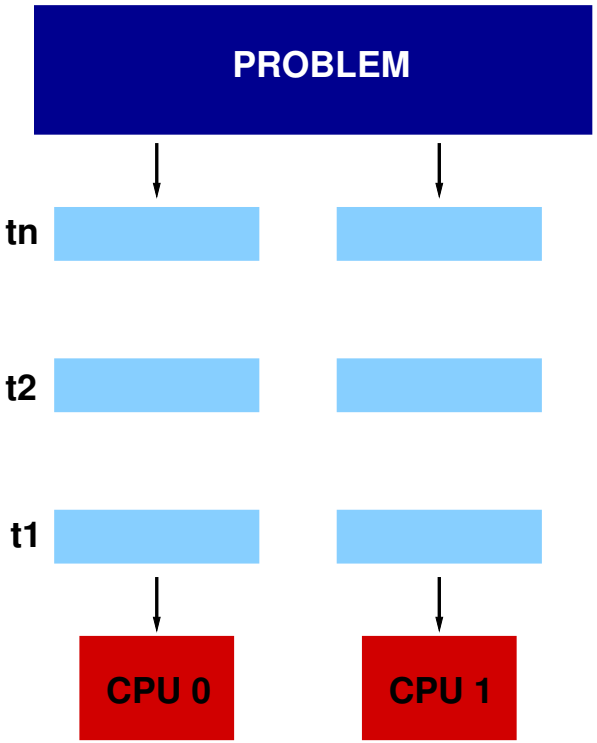
- memory restrictions
- execution speed
- cost of components



What is parallel computation?

Parallel computing

- problem statement
- translate problem into instructions
  some can be run *concurrently*
- use multiple processors
- execute instructions in *parallel*:
  more than one at a time



Reasons for using parallel processing

Principal reasons

- save *wall-clock time* (not CPU time!)
- solve larger problems (memory constraints)
- solve different related problems at the same time

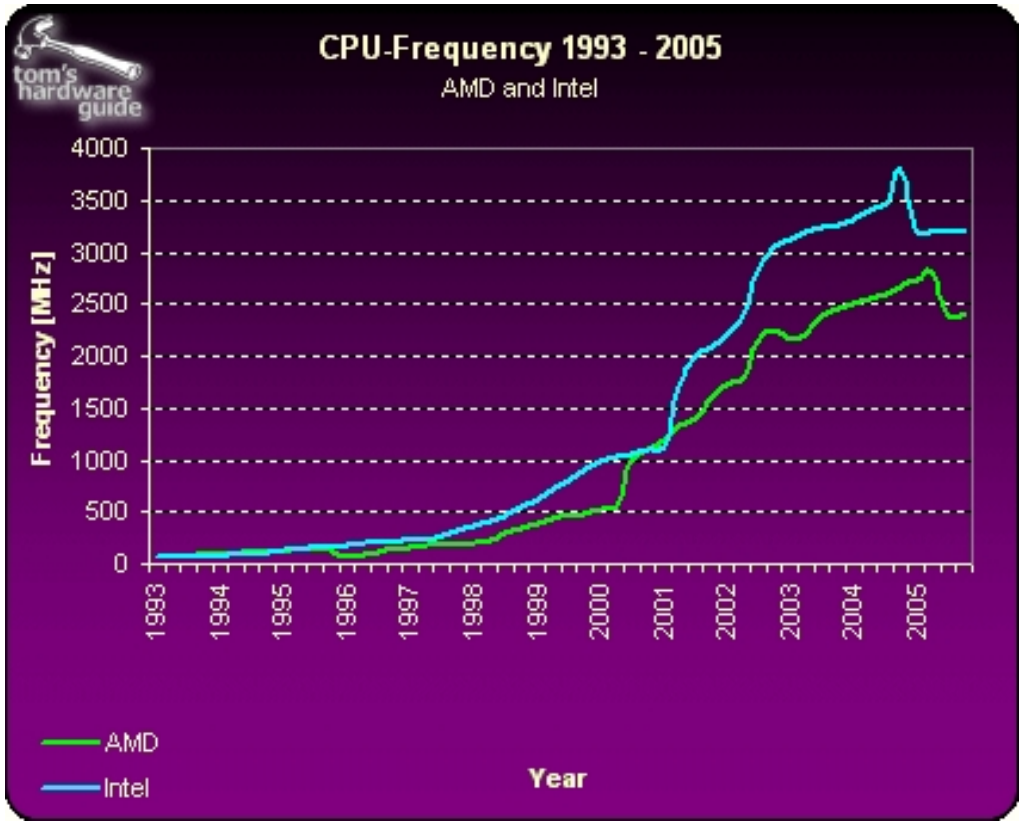
Other reasons

- availability of multiple low-cost resources
- simultaneous use of various non-local machines

Development of single-processor performance

Bottlenecks:

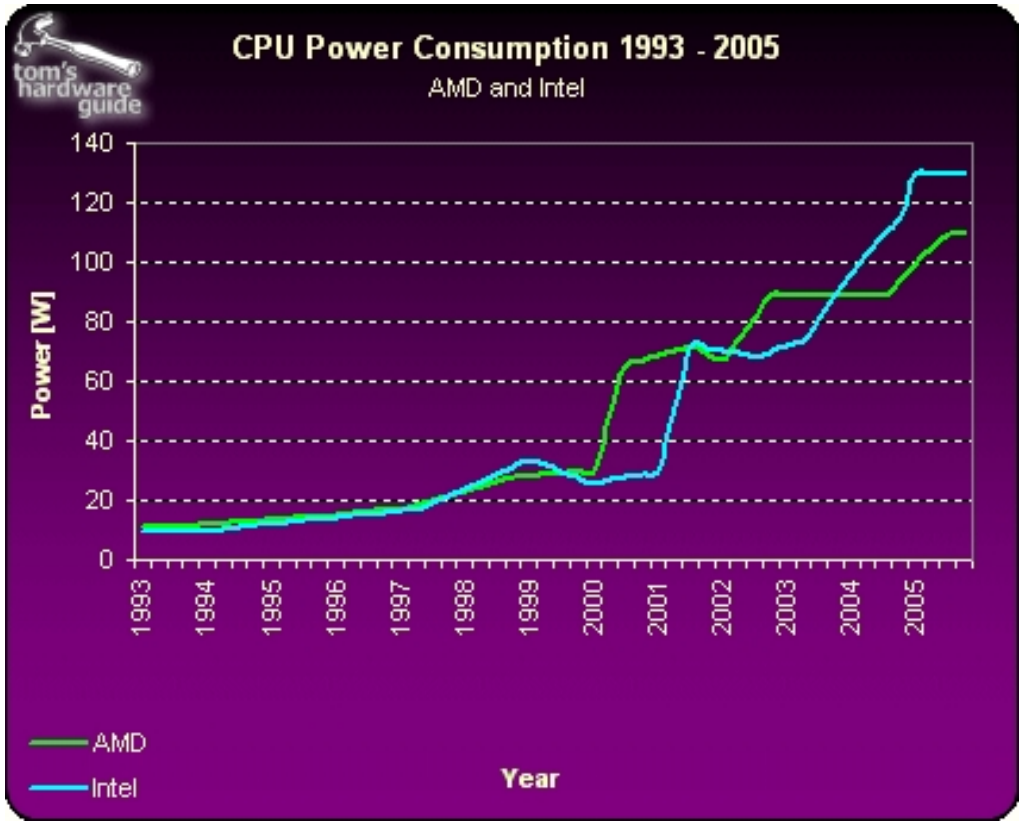
- frequency (power, heat)
- cache size
- execution flow



Development of single-processor performance

Bottlenecks:

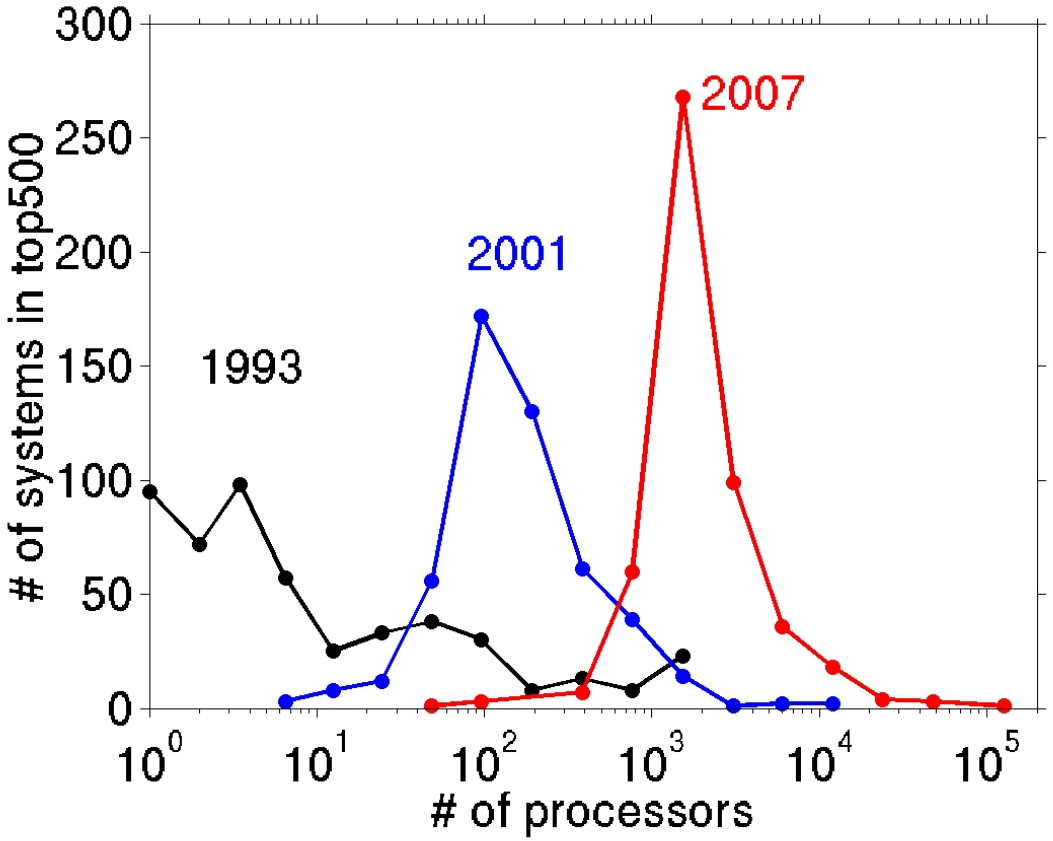
- frequency (power, heat)
- cache size
- execution flow



Development of multi-processor performance

Development of multi-processor performance

Number of processors per supercomputer



Development of multi-processor performance

Trend:

- increasing number of processors per system

$\Rightarrow$ parallel programming required!

Parallel computer architectures

Von Neumann architecture

Flynn's taxonomy

- data/instruction multiplicity

Memory architectures

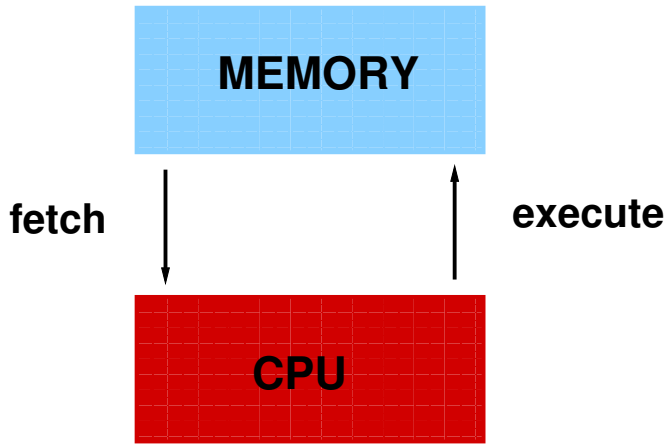
- locality of CPU/memory relation

Von Neumann architecture

single CPU:

- fetches instructions & data
- operates on data
- writes results to memory

$\Rightarrow$ serial, scalar computer
(most non-supercomputers)

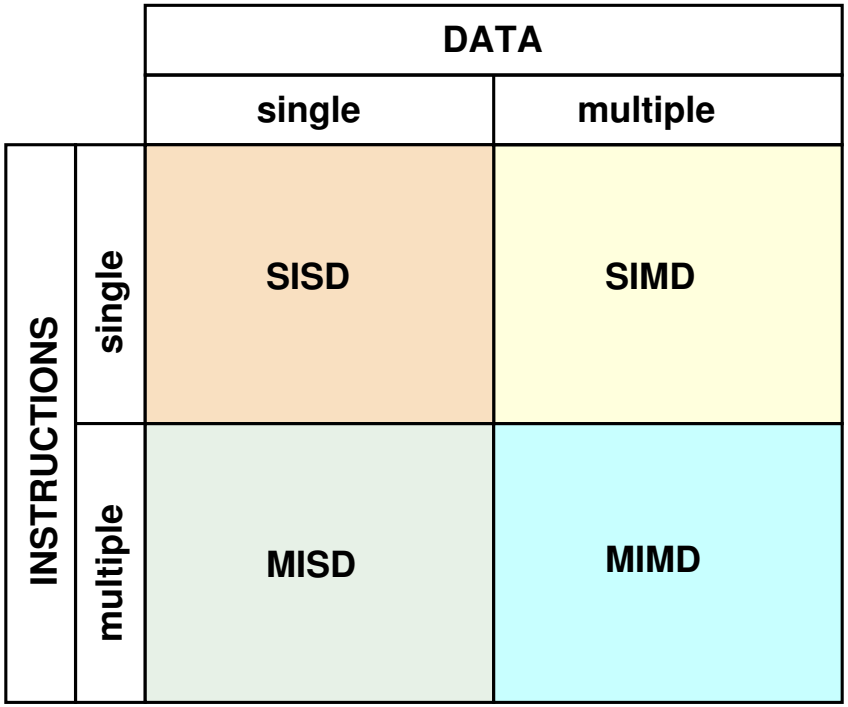


Flynn's taxonomy

Criteria:

- single/multiple instructions
- single/multiple data

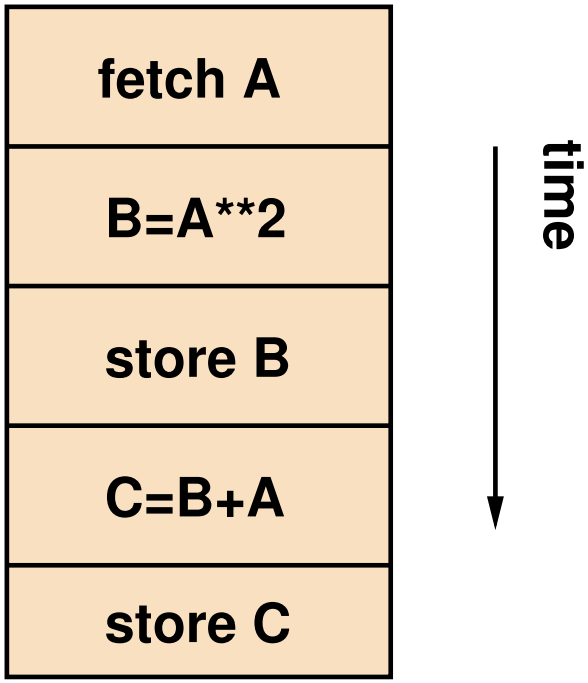
$\Rightarrow$ 4 possible combinations



Single Instruction Single Data (SISD)

- serial computer
- one instruction stream
- one data stream

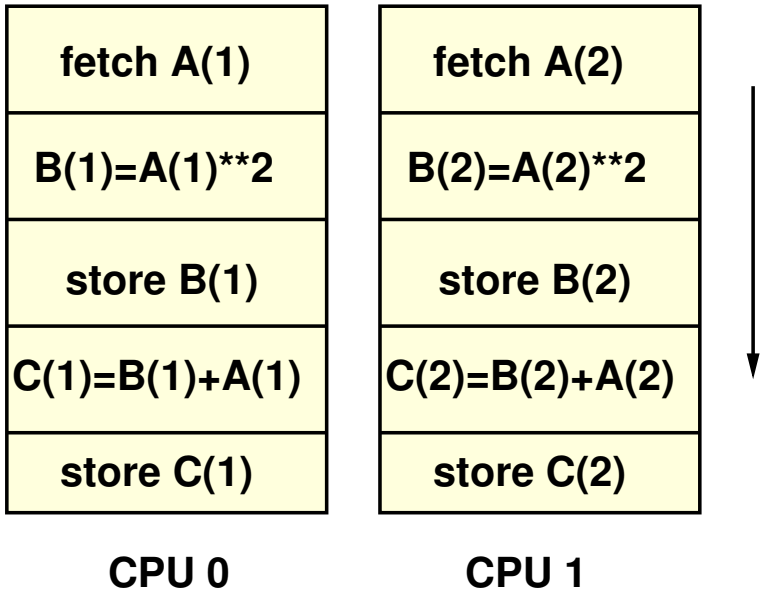
$\Rightarrow$ most (older) PCs



Single Instruction Multiple Data (SIMD)

- one instruction stream
- multiple data streams
- synchronous execution
- suited for regular problems (long loops)

$\Rightarrow$ processor/vector array
Cray C90, Fujitsu VP



Multiple Instruction Single Data (MISD)

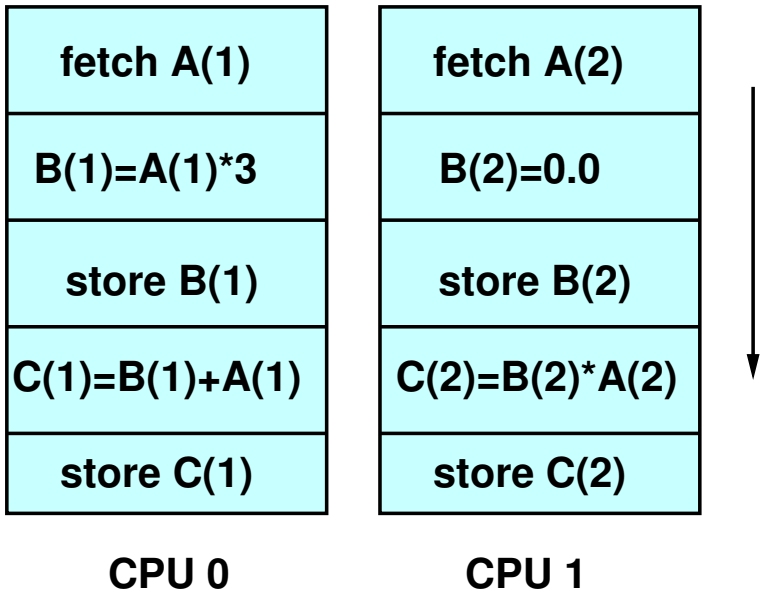
- multiple instruction streams
- one data stream

$\Rightarrow$ no machines of this type have been built
(apply different filters to single data?)

Multiple Instruction Multiple Data (MIMD)

- multiple instruction streams
- multiple data streams
- synchronous or asynchronous

$\Rightarrow$ most modern computers (SMPs, PC clusters)



Memory architectures

Locality of CPU/memory relation

Types of architectures:

- shared memory
- distributed memory
- hybrid

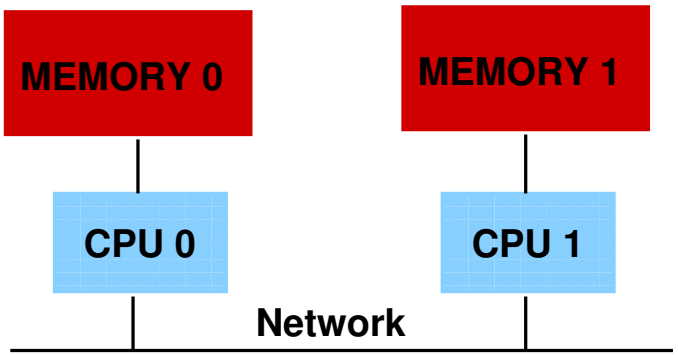
Memory location $\leftrightarrow$ Communication

Shared Memory

All processors have direct access to the same memory

- advantages: relatively easy to use, fast "communication"
- disadvantages: expensive to scale to large # CPUs, memory bandwidth bottleneck, cache coherency

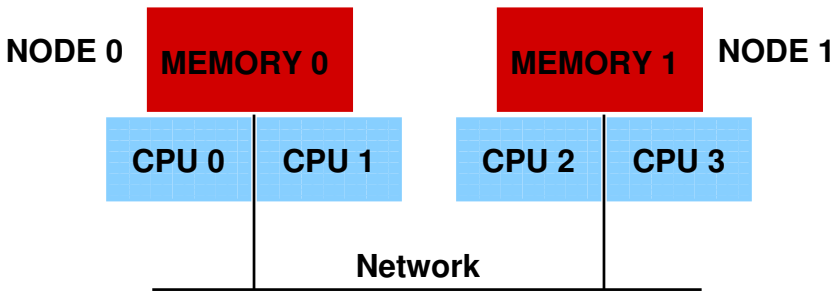
Distributed Memory



Each CPU has its own memory – communicate via network

- advantages: memory scales with # CPUs, no memory bandwidth bottleneck, no cache coherency problem
- disadvantages: more complex programming models, costly network hardware, network latency/bandwidth

Hybrid Shared/Distributed Memory



Nodes with shared memory – interconnected via network

- advantages & disadvantages of both types
- today most large clusters are of this type

Hardware examples (1)

Commodity PC cluster

- distributed memory
- commodity hardware (PCs, Gbit ethernet)
- open source software (GNU/Linux OS, MPICH/LAM)

Hardware examples (2)

Integrated PC cluster

- distributed/shared memory
- semi-specialized hardware (blade PCs, Gbit ethernet/fiberoptics)
- open source software (GNU/Linux OS, MPICH/LAM)

Hardware examples (3)

MareNostrum, BSC (# 13)

- hybrid: shared/distributed memory
- 2 dual-core Power PCs per node, Myrinet network
- GNU/Linux OS, IBM compiler, proprietary MPI

Hardware examples (4)

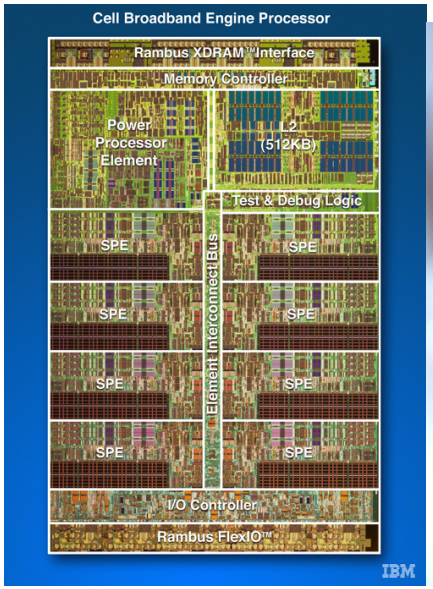
IBM BlueGene (# 1)

- hybrid: shared/distributed memory
- 2 Power PCs per node, triple network
- low FLOPS per Watt
- dense packing of nodes
  (LLNL: 100496 nodes)

Hardware examples (5)

Cell processor (Playstation 3)

- shared memory
- 1 Power Processor
  + 8 *Synergistic Processing Elements*
- still experimental!



Parallel programming models ('paradigms')

Serve as abstraction between hardware and application

Principal models which are currently in use:

- parallel threads (OpenMP)
- message passing (MPI)
- data parallel (High Performance Fortran)
- hybrid models (OpenMP & MPI)

Choice depends on application

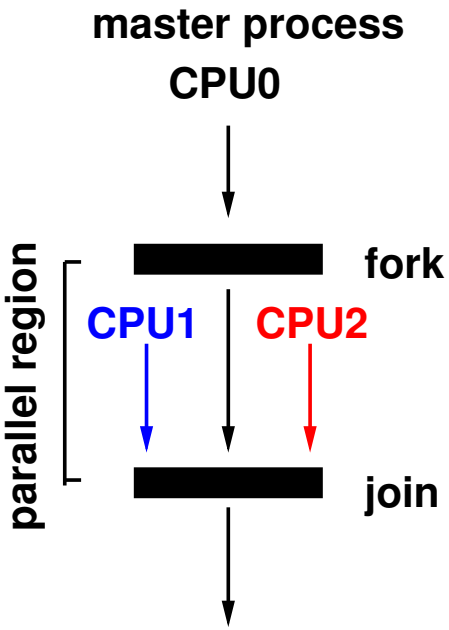
Threads model – (OpenMP)

Characteristics:

- suited for shared-memory machines
- additional threads created in time, working on parallel regions
- based upon compiler directives

$\Rightarrow$ ease of transition from serial programs

$\rightsquigarrow$ not suited for distributed memory!



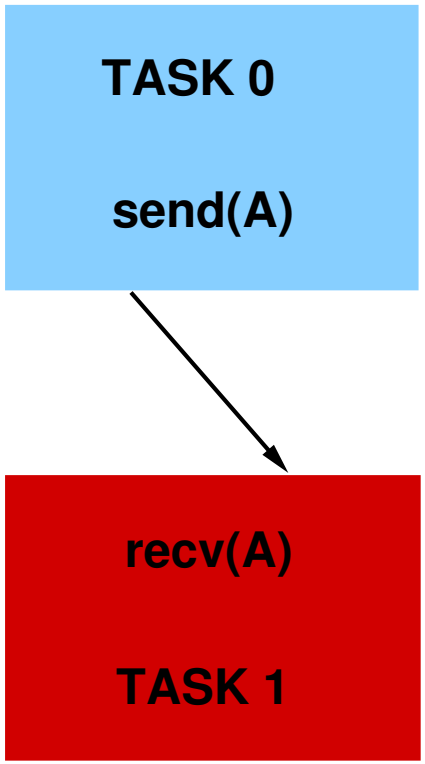
Message passing model – (MPI)

Characteristics:

- suited for all memory architectures
- explicit, co-operative communication
- based upon library calls

$\Rightarrow$ full control over communication

$\rightsquigarrow$ more complex programming



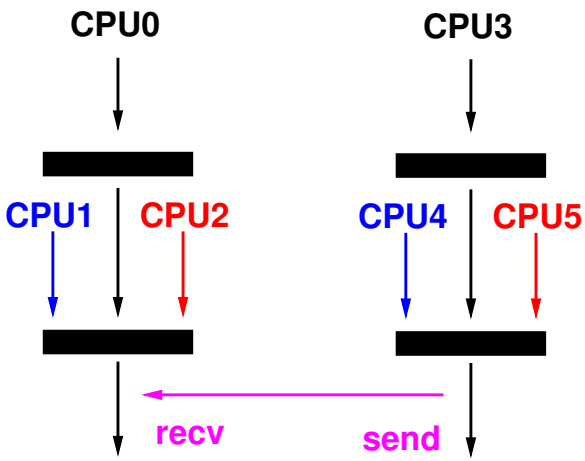
Hybrid model – (MPI & OpenMP)

Characteristics:

- suited for hybrid architectures: multi-processor nodes & network
- threads inside shared-memory nodes
- explicit communication between nodes

$\Rightarrow$ possibly improves performance

$\rightsquigarrow$ even more complex programming



Design of parallel programs

When to use parallel computation?

- problem does not fit into single-processor memory
- need the answer faster

When NOT to use parallel computation?

- problem is inherently sequential
  e.g.: recurrent series $f(k) = a \cdot f(k-1) + b \cdot f(k-2)$
- problem is naturally parallel
  e.g.: independent sweeping of parameter space

Types of parallel algorithms

Embarassingly parallel

- no communication needed
- when some post-processing is needed: "parametric"

  e.g.: run the same model, different initial conditions

Data parallel

- distribution of input data to processors,
  local operations without communication
- some global post-processing operation

  e.g.: parallel search algorithm

Synchronous (or: tightly coupled)

Some intensive parallel computing applications

Quantum mechanics

- Physics, Chemistry

Molecular dynamics

- Chemistry, Biology, Materials science

Fluid dynamics

- Astronomy, Engineering, Weather/climate research

Financial analysis

...

Design strategy for parallel programs

Step by step

1. Identify concurrent parts of the algorithm
2. Choose a problem decomposition
3. Design the communication patterns
4. Implement, debug & optimize

$\rightsquigarrow$ iterative procedure!

Identify concurrent parts of the algorithm

Consider different solution strategies

- most efficient sequential method not always optimal in parallel
- be ready to modify underlying numerics
- example
  PDE discretization: spectral $\longrightarrow$ finite-differences ? fluid
  flow problem: Navier-Stokes $\longrightarrow$ Lattice-Boltzmann ?

Identify hotspots where the main work is done

- true benefit from parallelism

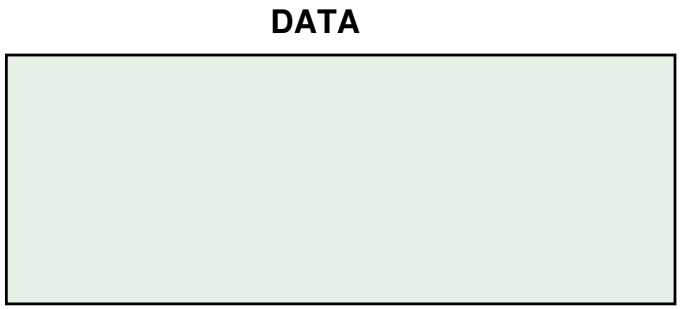
Identify bottlenecks which inhibit parallelism

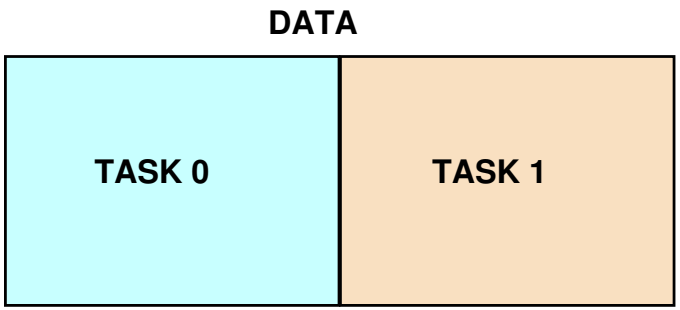
- I/O operations, data dependencies, ...

Domain decomposition or functional decomposition?

Domain decomposition

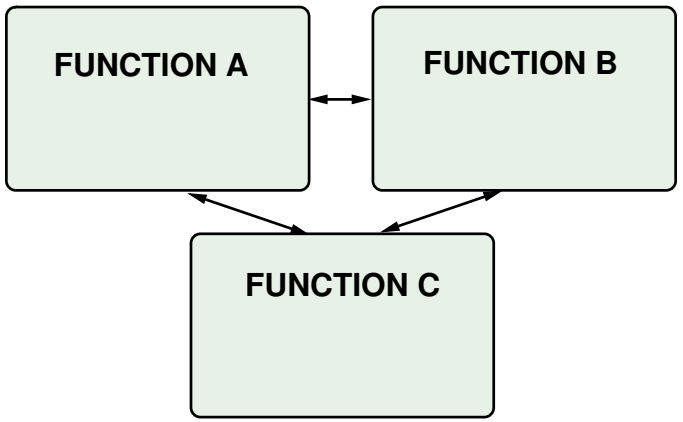
- each task works on its part of the data (e.g. block of a matrix)





Functional decomposition

- each task treats different parts of the algorithm (e.g. coupled models)

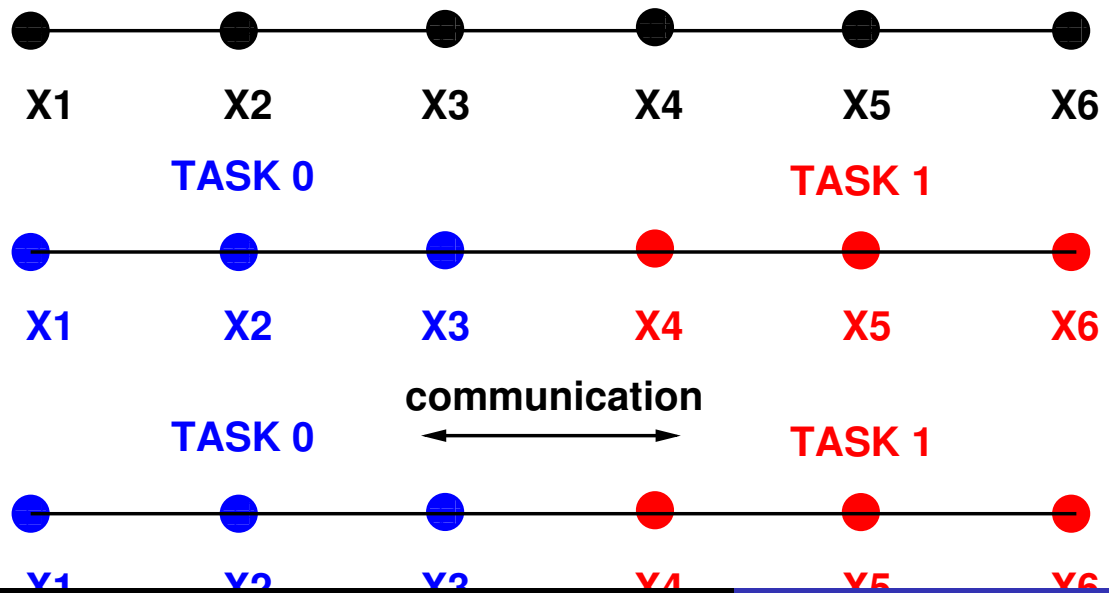


Why do tasks need to communicate?

Data dependency

Example:

- solving partial differential equation $\partial_t u = a \cdot \partial_x u$
- finite-differences & domain decomposition



Cost of communication

Latency

- time it takes to initiate communication
  example: BlueGene $\approx 3.5\mu s$; InfiniBand $\approx 1.3\mu s$

Bandwidth

- amount of data which can be communicated per time
  example: BlueGene $\approx 5 Gbit/s$; InfiniBand $\approx 10 Gbit/s$

In general: avoid small messages!

Communication patterns

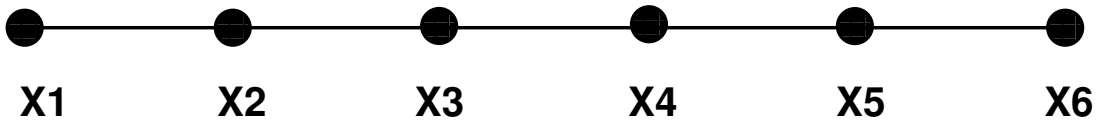
Points to be considered:

- which data needs to be transferred?
- when?
- which tasks participate?
- ordering of communication
- synchronous or asynchronous communication?
- point-to-point vs. global communication
- type of memory architecture

Communication patterns

Example

- PDE: $\partial_t u = a \cdot \partial_x u$
- discretized: $u_i^{n+1} = u_i^n \cdot (1 + a\Delta t/\Delta x) - u_{i-1}^n \cdot a\Delta t/\Delta x$



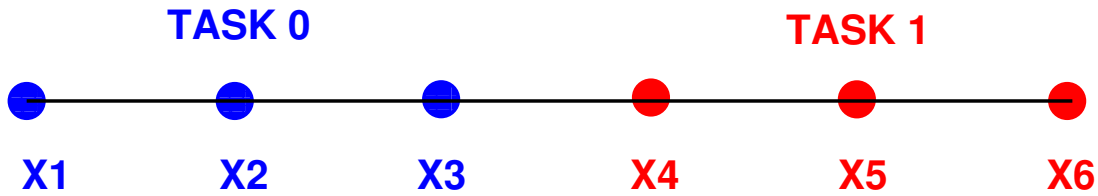
- sequential algorithm:

```
for each time step
  for i=1:N
    v(i)=u(i)*(1+A)-u(i-1)*A
  end
  u=v
end
```

Communication patterns

Example continued

- domain decomposition:



- communication: u_3^{n+1}, task 0 $\longrightarrow$ task 1, end of each step

Communication patterns

Parallel algorithm:

```
for each time step
  communicate:  u(3), task 0 → task 1
```

```
task 0
for i=1:3
  v(i)=u(i)*(1+A)-u(i-1)*A
end
u=v
```

```
task 1
for i=4:6
  v(i)=u(i)*(1+A)-u(i-1)*A
end
u=v
```

```
end
```

Other considerations concerning the design

Load balancing

- even distribution of work among tasks

  example: fluid-particle motion

Coding complexity

- development time and effort of parallel programs

Portability

- guarantees flexibility at the time of execution

Performance measures

Parallel speedup

- $S \equiv \frac{t_{exec}(1)}{t_{exec}(np)}$
- ideally an application has $S = np$

Parallel efficiency

- $E \equiv \frac{S}{np}$
- ideally an application has $E = 1$

Limits of parallel performance

Amdahl's law

- $S \leq \frac{1}{\frac{F_{par}}{np} + F_{ser}}$ $\qquad \lim_{np \to \infty} S = \frac{1}{1 - F_{par}}$
- potential speedup limited by parallel fraction of code F_{par}

Scalability

- ability of maintaining parallel efficiency WHILE increasing the problem size AND adding processors
- example $\rightarrow$ good scalability

# grid points	np	t_{exec}
10^6	10	$1.00s$
$2 \cdot 10^6$	20	$1.01s$

Parallel input/output

I/O is in general inhibitor of parallelism

- parallel i/o file systems are not always available
- simultaneous writing can lead to overwriting

$\Rightarrow$ have (a) dedicated task(s) assigned to i/o operations

When local disc space is available:

- non-local i/o (over network) is very slow

$\rightsquigarrow$ avoid as much as possible

$\Rightarrow$ each task writes its own data to a separate file

Parallel code debugging

Use write statements

- tedious, but often unavoidable

Compare sequential vs. parallel version

- useful to have a working sequential code as starting point

Existing parallel debugging tools

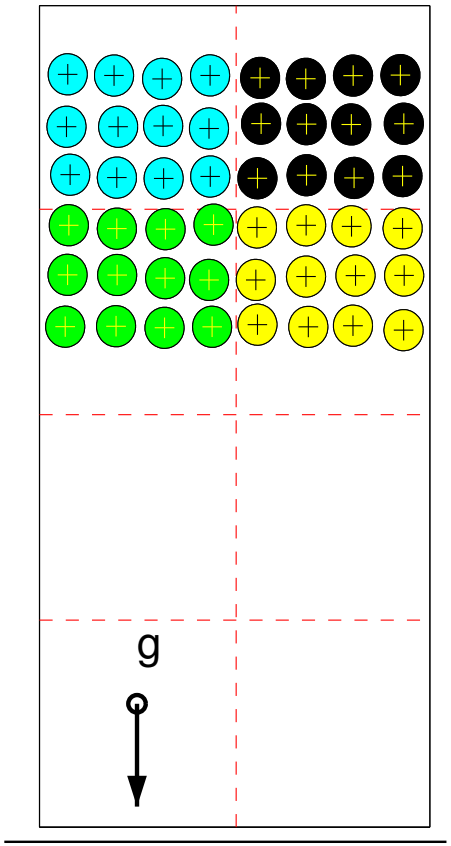
- console: pdbx
- GUI: totalview, xmpi

Acknowledgement

This lecture heavily draws from the following resources:

- https://computing.llnl.gov/tutorials/parallel_comp
- http://www.mhpcc.edu/training/workshop/parallel_intro

Example application



- incompressible Navier-Stokes + 48 rigid particles
- 2×4 processors
- color indicates "master" processor
- iso-contours show streamlines ($\pm$)

$\rightsquigarrow$ particle work is unbalanced!

Another example

Array processing

- data parallel algorithm
- "pool of tasks" technique

$\Rightarrow$ granularity