Parallel and Distributed Programming: foundations

Fernando Silva

Computer Science Department
Center for Research in Advanced Computing Systems (CRACS)
University of Porto, School of Sciences
http://www.dcc.fc.up.pt/~fds

Agenda

1. **Introduction and Concepts**
   - Motivation for Parallelism
   - Defining Parallelism
   - Parallel Architectures
   - Parallel Programming (overview)
     - ★ design steps
     - ★ programming models and paradigms
     - ★ small example

Why should we go parallel? (1)

The scenario:

Given a problem P, our best sequential algorithm can solve it in N time units using 1 processor.

Ideal solution:

Solve the same problem in 1 time unit using N processors at same time (i.e. in parallel).

Why should we go parallel? (2)

- **Two main reasons to use parallelism:**
  - reduce the execution time needed to solve a problem
  - be able to solve larger and more complex problems
- Other important reasons:
  - computing resources became a commodity and are frequently under-utilized
  - overcome the physical limitations in chip density and production costs of faster sequential computers
  - overcome memory limitations as the solution to some problems require more memory then one could find in just one computer

Simulation: the third pillar of science

(slide from Kathy Yelick, CS267 lecture 1)

- Traditional scientific and engineering paradigm:
  1. do theory or paper design
  2. perform experiments or build system.
- Limitations
  - too difficult - build large wind tunnels
  - too expensive - build a throw-away passenger jet
  - too slow - wait for climate or galactic evolution
  - too dangerous - weapons, drug design, climate experimentation
- Computational science paradigm:
  3. use high-performance computer systems to simulate the phenomenon
     - ⋆ based on known physical laws and efficient numerical methods

Grand Challenge Problems

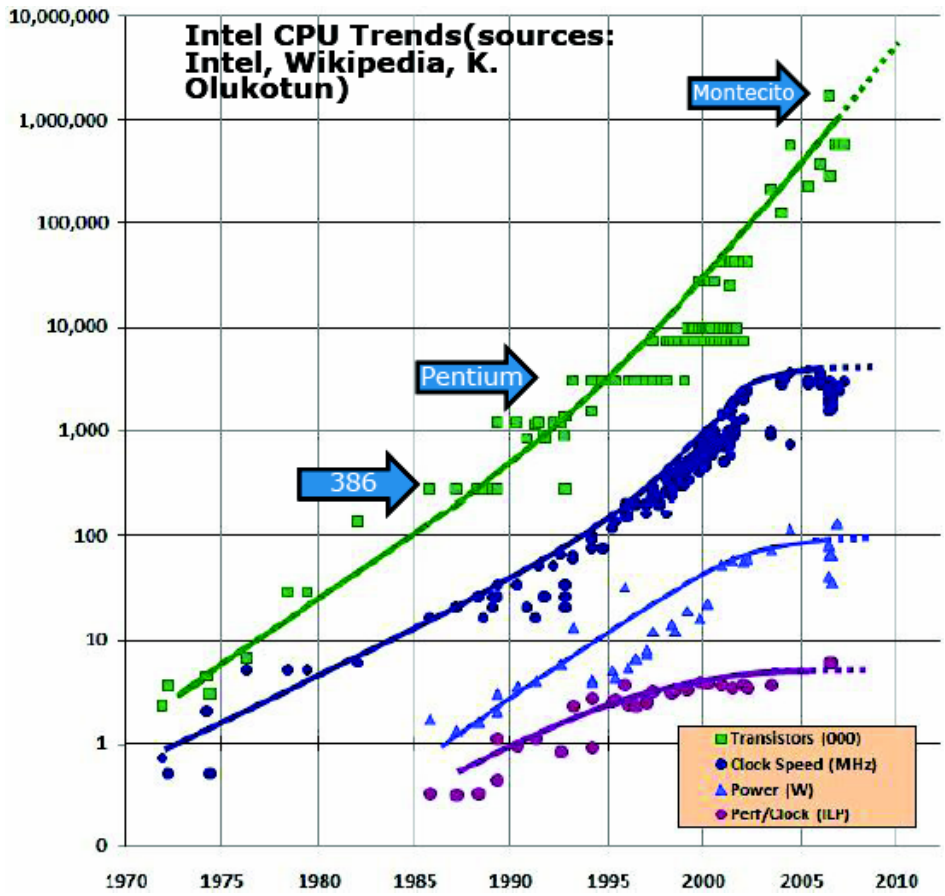
- The solution or simulation of fundamental problems in science and engineering, with a strong scientific and economic impact, known as Grand Challenge Problems (GCPs), have been the driving force for Parallel Computing.
- Typically, GCPs simulate phenomena that cannot be measured by experimentation:
  - Global Climate modeling
  - Biology: genomics; protein folding, drug design
  - Astrophysical modeling
  - Computational Chemistry
  - Earthquake and structural modeling
  - Computational fluid dynamics (airplane design)
  - Crash simulation
  - Financial and economic modeling

New Data-Intensive Applications

- Currently, large volumes of data data are produced and their processing and analysis also require high performance computing.
  Some examples:
  - Data mining
  - Web search
  - networked video
  - Video games and virtual reality
  - Computer aided medical diagnosis
  - ...
- Similarly, data is collected and stored at enormous speeds (GByte/hour).
  Some examples:
  - sensor data streams
  - telescope scanning the skies
  - micro-arrays generating gene expression data

Free Lunch is Over (Herb Sutter, 2005)

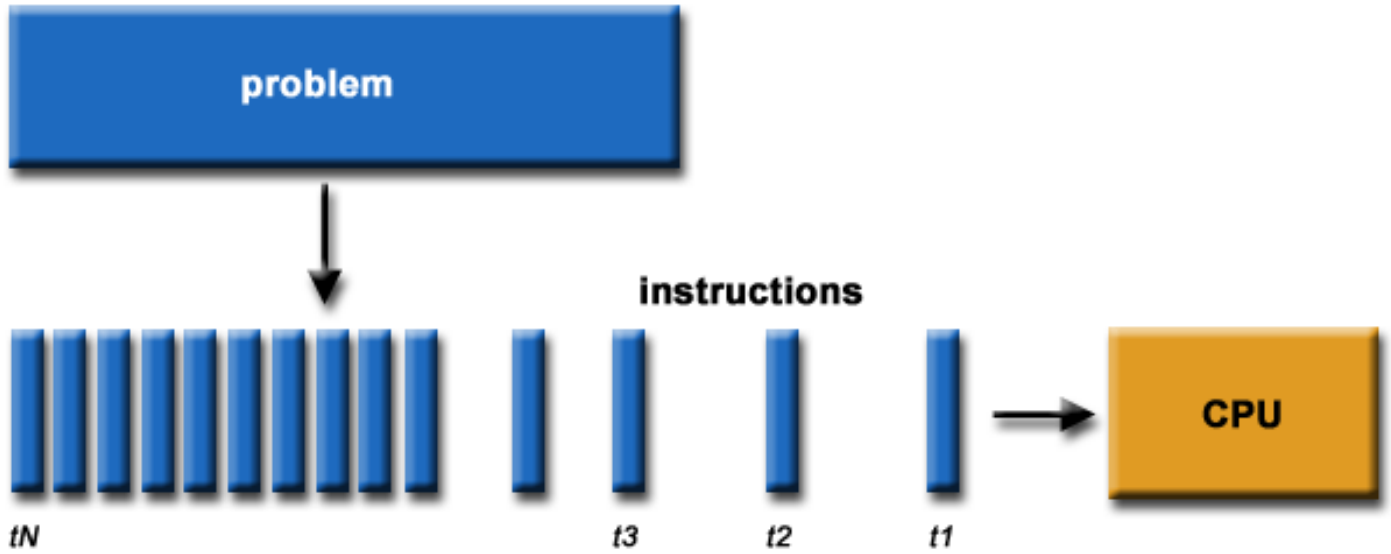
- Chip density still increasing $\sim 2\times$ every 2 years, but
  - production is very costly
  - clock speeds hit the wall
  - heat dissipation / cooling problems
- Chips already integrate many parallel ideas:
  - super-scale execution,
  - super pipelining
  - branch prediction
  - out-of-order execution
  - speculative execution



- Manufacturer's solution: multiple cores on the same chip, i.e. go for parallelism.

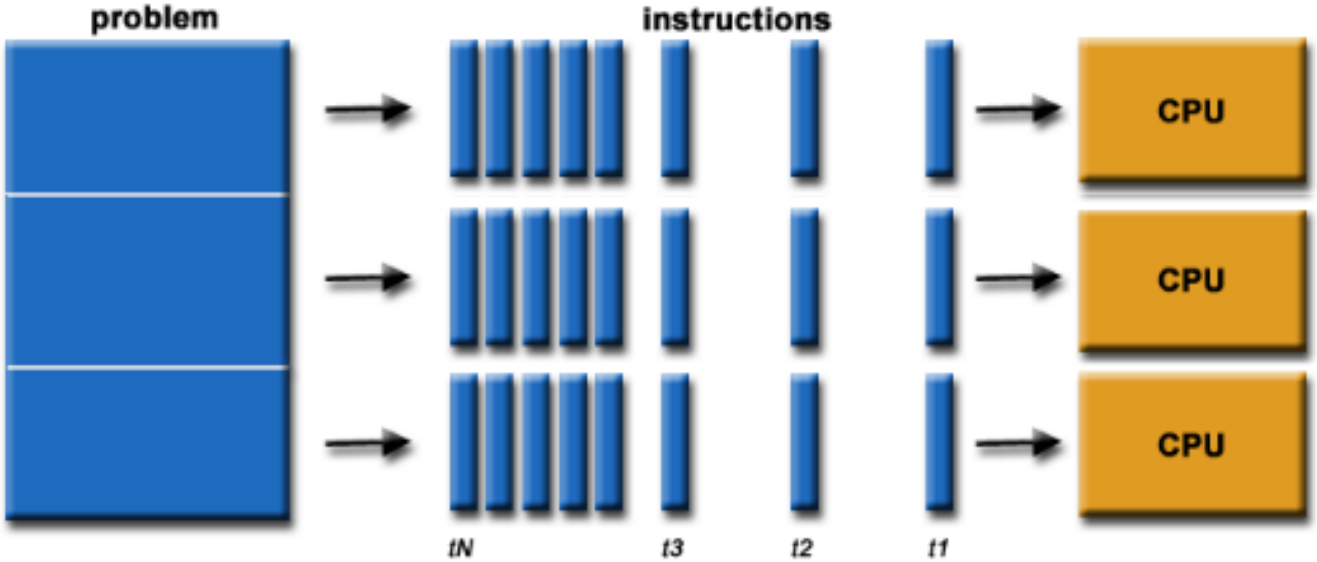
Sequential Computing

- Solve a problem by executing one flow of execution using one processor (CPU).



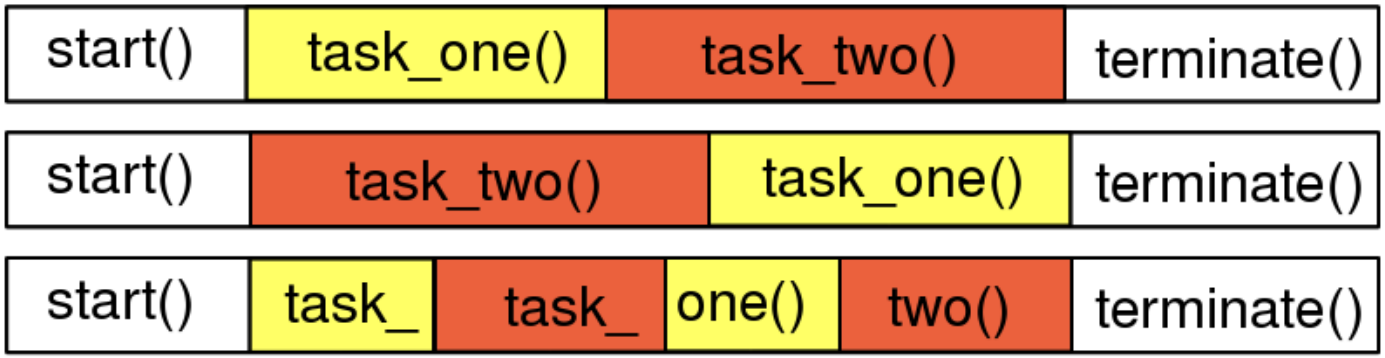
Parallel Computing

- Decompose your problem in multiple parts that can be executed concurrently, and
- have multiple execution flows, one for each part, using multiple processors (CPUs) simultaneously.
- Potentially, execution time should decrease.



Concurrency or Potential Parallelism

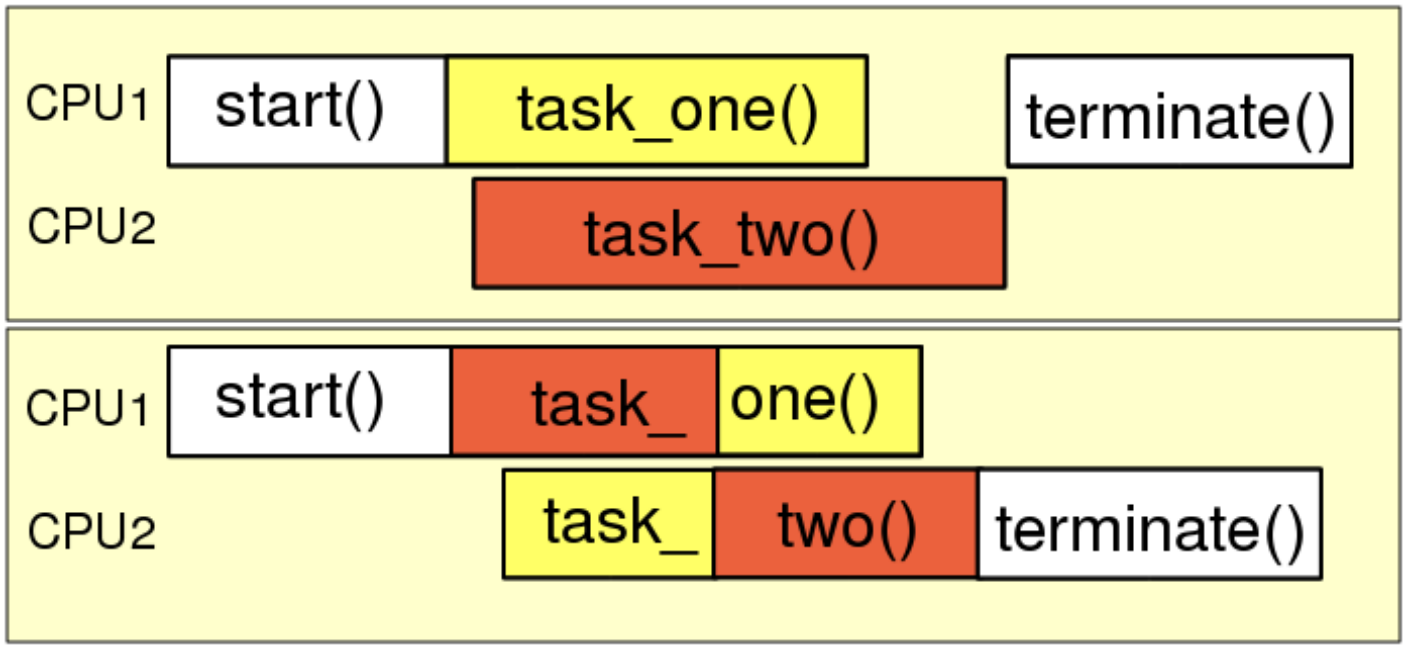
- A program exhibits concurrency (or potential parallelism) when it includes tasks (parts of the program) that can be executed in any order without changing the expected result.
- An illustration of different arrangements for execution:



- Smaller tasks, if not too small, simplify possible arrangements for execution.
- Proportion of sequential tasks to start and terminate execution should be small as compared to the parts that are concurrent.

Parallelism

- Parallelism is exploited when the tasks of a program are executed simultaneously in more then one processing unit.
- An illustration of two possible arrangements for execution are:



Implicit Parallelism

- Parallelism is exploited implicitly when it is the compiler and runtime system that:
  - automatically detects potential parallelism in the program
  - assigns the tasks for parallel execution
  - controls and synchronizes execution
- (+) frees the programmer from the details of parallel execution
- (+) it is a more general and flexible solution
- (-) very hard to achieve an efficient solution for many applications

Explicit Parallelism

- Parallelism is exploited explicitly when it is left to the programmer to:
  - annotate the tasks for parallel execution
  - assign tasks to processors
  - control the execution and the synchronization points
- (+) experienced programmers achieve very efficient solutions for specific problems
- (-) programmers are responsible for all details of execution,
- (-) very hard to achieve an efficient solution for many applications
- (-) programmers must have deep knowledge of the computer architecture do achieve maximum performance.

Parallel Computing

Putting it simply, we can define

Parallel Computing

as the use of multiple computing resources to reduce the execution time required to solve a given problem.

These computational resources can include:

- multiprocessors: one machine with multiple processors
- multicomputers: an arbitrary number of dedicated interconnected machines
- clusters of multiprocessors: a combination of the above

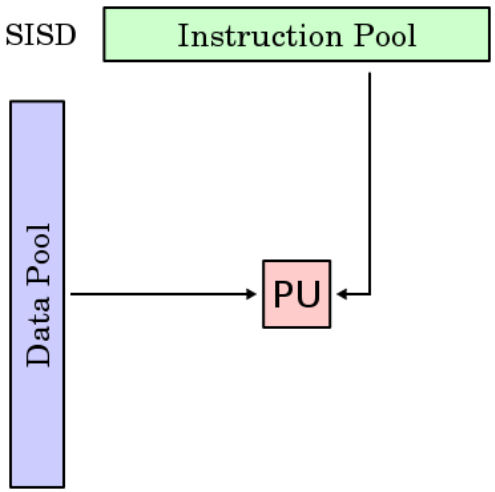
Flynn Taxonomy

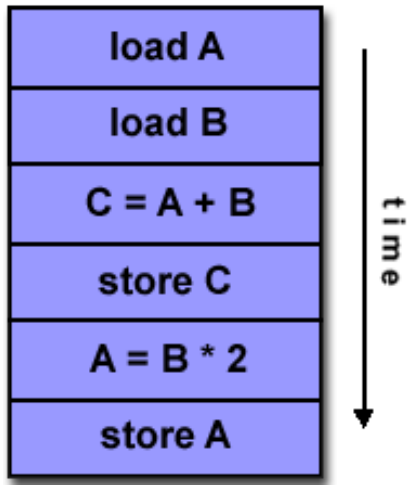
- Flynn (1966) proposed a taxonomy to classify computer architectures.
- Analyzes two independent dimensions available in the architecture:
  - number of concurrent instructions (or control)
  - number of concurrent data streams

	Single Data	Multiple Data
Single Instruction	**SISD** *Single Instruction Single Data*	**SIMD** *Single Instruction Multiple Data*
Multiple Instruction	**MISD** *Multiple Instruction Single Data*	**MIMD** *Multiple Instruction Multiple Data*

SISD - Single Instruction Single Data

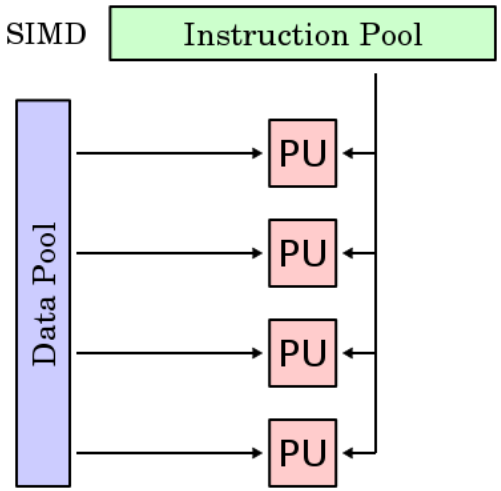
- Exploits no parallelism; corresponds to sequential architectures
  - only one instruction is processed at a time
  - only one flow of data is processed at a time
- Examples: PCs, workstations and servers with one CPU.

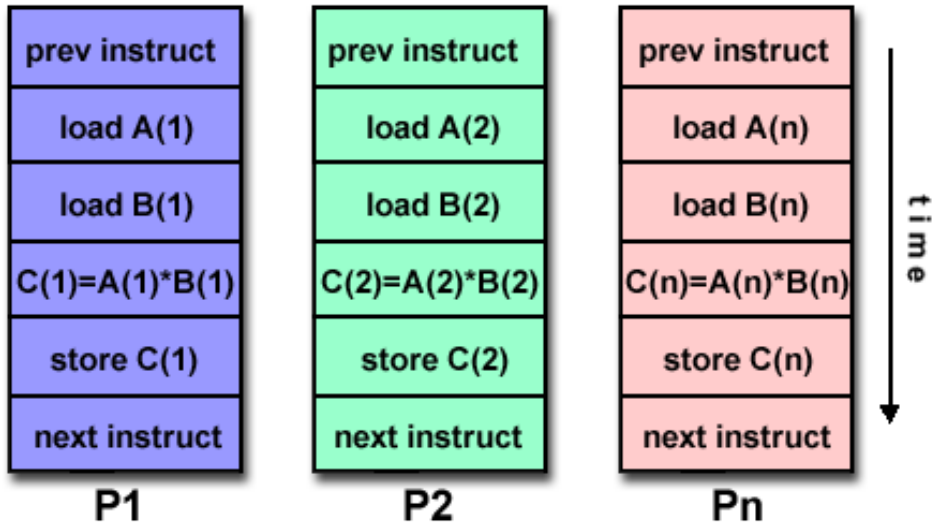




SIMD - Single Instruction Multiple Data

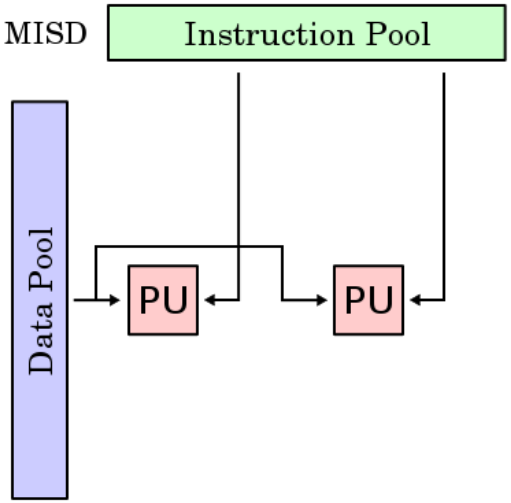
- Parallel architecture specifically designed for problems characterized by high regularity in the data (e.g. image processing)
  - all processing units execute the same instruction at each time
  - each processing unit operates on a different data stream
- Examples: array processors, graphics processing units (GPUs)

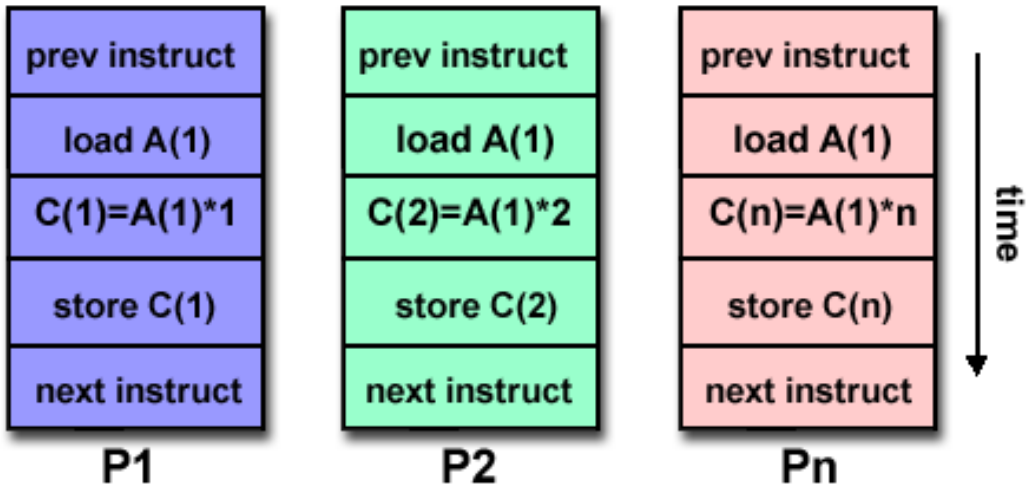




MISD - Multiple Instruction Single Data

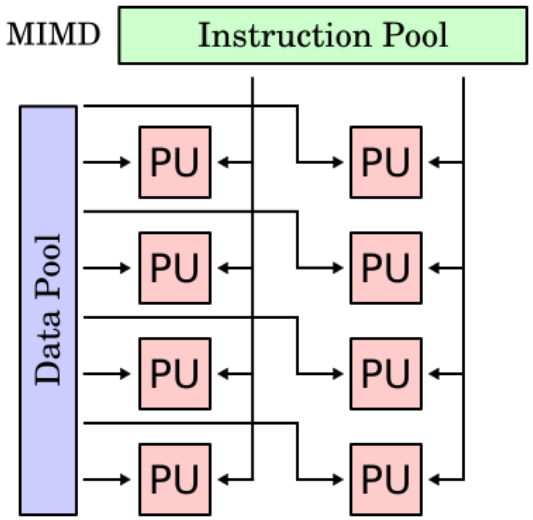
- Uncommon parallel architecture where each processing unit performs a function on the same data stream.
  - each processing unit executes different instructions at each time
  - the processing units operate on the same data stream, trying to agree on the result (common for control) or by operating in a pipeline fashion.
- Examples: fault tolerance computers, systolic arrays (arguably).

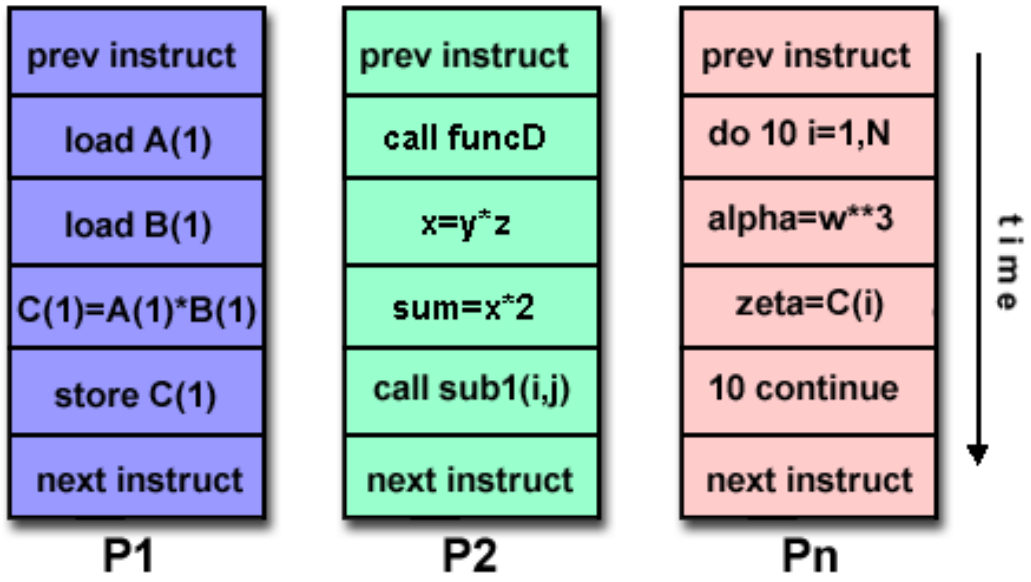




MIMD - Multiple Instruction Multiple Data

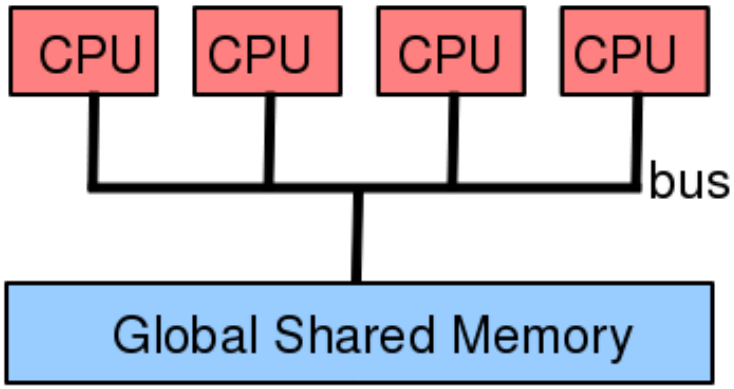
- The most common parallel architecture.
  - each processing unit executes different instructions at each time
  - each processing unit can operate on a different data stream
- Examples: multiprocessors (now multicores), multicomputers (also clusters)





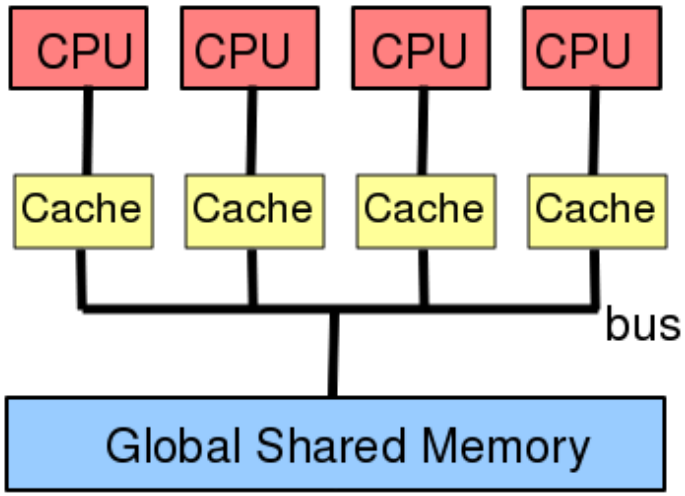
Multiprocessors or Shared Memory Machines

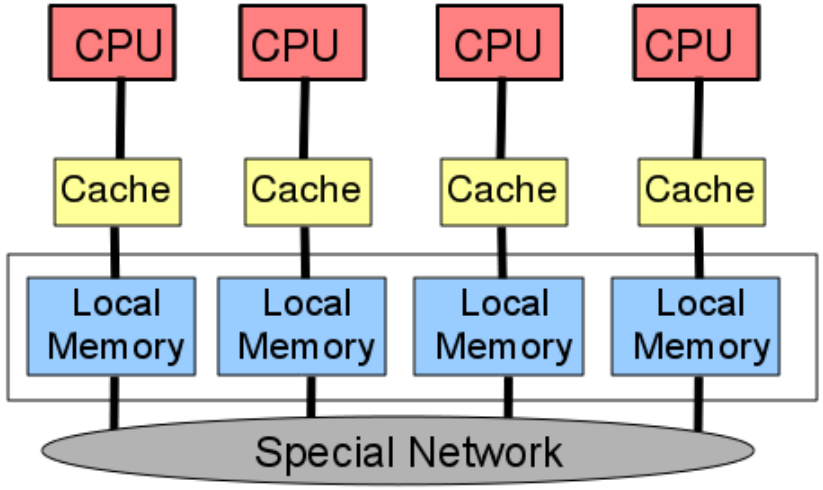
- A multiprocessor or a shared memory machine is parallel computer in which all processors share the same physical memory.
  - processors execute independently but share a global address space
  - any modification on a memory position by a processor is equally viewed by all other processors.
- bus congestion imposes limits to scalability
- including a cache between each processor and memory helps.



Classes of Multiprocessors

- Uniform Memory Access (UMA)
  - equal cost to access memory
  - cache consistency implemented in hardware (write invalidate protocol)
- Non-Uniform Memory Access (NUMA)
  - different access times to different memory regions
  - cache consistency implemented in hardware (directory-based protocol)





Multiprocessors: advantages and disadvantages

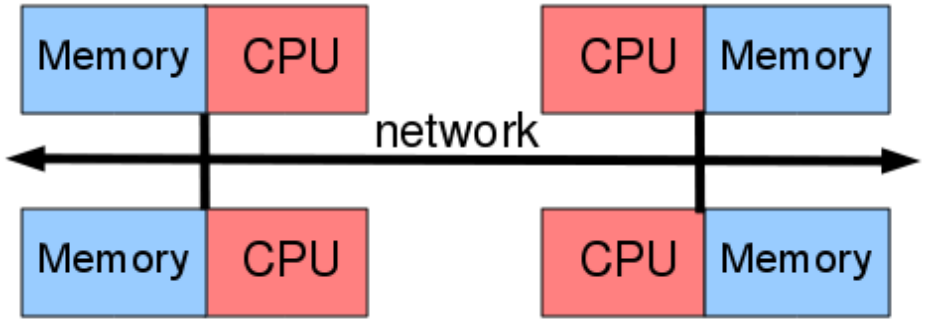
- (+) Data sharing among concurrent tasks is simple, uniform and fast.
- (+) Simpler programming model as there is a global view of memory
- (-) Requires synchronization mechanisms to modify shared data (normally locks)
- (-) Not scalable. Increasing the number of processors, increases bus congestion to access memory and thus makes cache coherency mechanisms impractical
- (-) High cost, specially due to very expensive bus (or interconnect network) and caches

BUT, the disadvantages are being now overcome with new designs and by bringing the multiprocessor into the chip (the multicores).

Notable machines: KSR-1, Sequent Symmetry, BBN Butterfly, CRAY Y-MP, ...

Multicomputers or Distributed Memory Machines

- A multicomputer or distributed memory machine is a parallel machine where each processor has its own local memory that is not directly accessible by other processors.
  - no shared memory, no global address space
  - each processor has its own address space
  - modifications by one processor in one position of memory are not visible by others
  - data sharing or synchronization takes place by exchanging messages



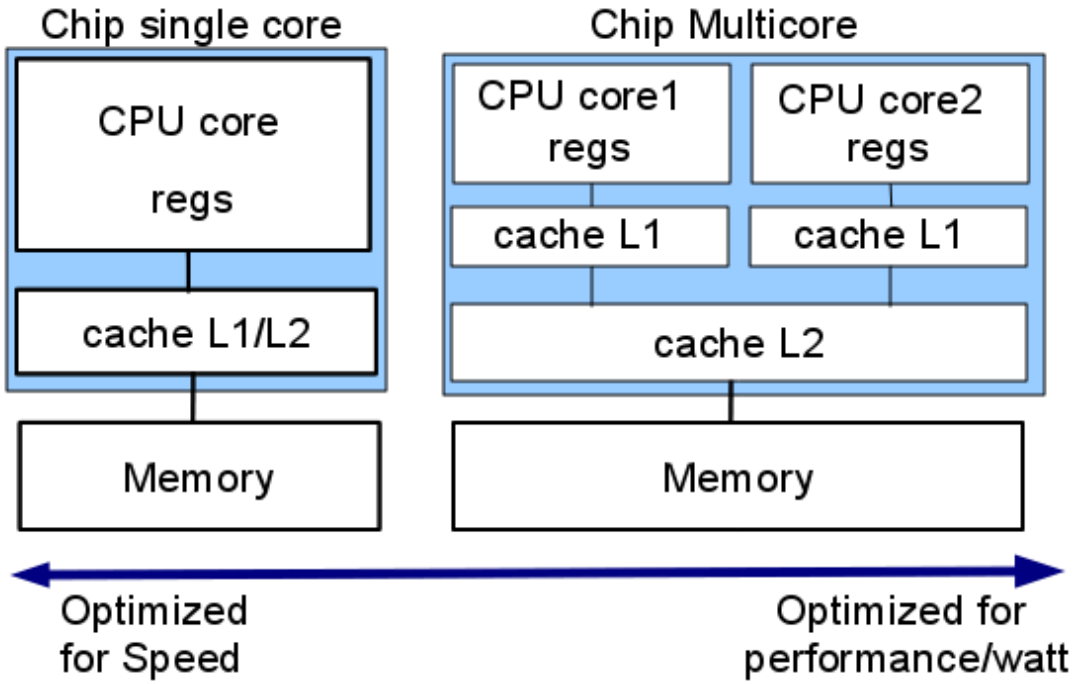
Multicomputers: advantages and disadvantages

- (+) high scalability on processors and memory
- (+) reduced cost, in fact they can be built using off-the-shelf components (beowulf cluster)
- (-) communication and synchronization via messages exchange only
- (-) non-remote data access is very costly in performance
- (-) harder to program, as the programmer has to control explicitly communication.

Notable machines: IBM SP2, Intel Paragon, CM-5, Parsytec GCPP, CRAY-3D,...

Recent Multiprocessors - the Multicore

- Multicore emphasize shared memory parallelism
  - reached mainstream desktops and game consoles
- Clusters of multiprocessors are now the norm:
  - 90% of the leading top500 supercomputers are based on multicore
  - leading to hybrid models of parallel programming



The Top500 Supercomputers List

This is a list updated twice a year with the fastest machines worlwide (peek performance in Tera-Flops) – July 2010:

Rank	Site	Model	Processor	Cores	Rmax
1	Oak Ridge	Cray XT5-HE	Opteron 6-core	224162	1759
2	Shenzhen (NSCS)	TC3600 Blade	Intel X5650	120640	1271
3	DOE/NNSA/LANL	BladeCenter/NVidia	Opteron	12240	1042
4	Univ. Tennessee	Cray XT5-HE	Opteron 6-core	98928	832
5	FZJ	IBM BlueGene/P	PowerPC	294912	825
6-US, 7-China, 8-US, 9-US. 10-US, 11-China, ..., 13-Russia					

Observe that:

- supercomputers nowadays are clusters of multicores
- number of cores exceeding 100,000 units

Architectures MIMD

We can summarize the MIMD architectures as follows:

	Multiprocessor		*Multicomputer*	
	CC-UMA	*CC-NUMA*	*Distributed*	*Distributed-Shared*
Scalability	10s CPUs	100s CPUs	1000s CPUs	
Communication	Shared Memory Segments *Threads* *OpenMP* (*MPI*)		*MPI*	*MPI/Threads* *MPI/OpenMP*

Parallel Programming (1)

We overviewed different types of parallel architectures, but the main question is how can we develop software that uses their full computing capacity?

the truth is that

parallel programming remains a very complex task

Parallel Programming (2)

An informal definition:

Parallel programming consists in

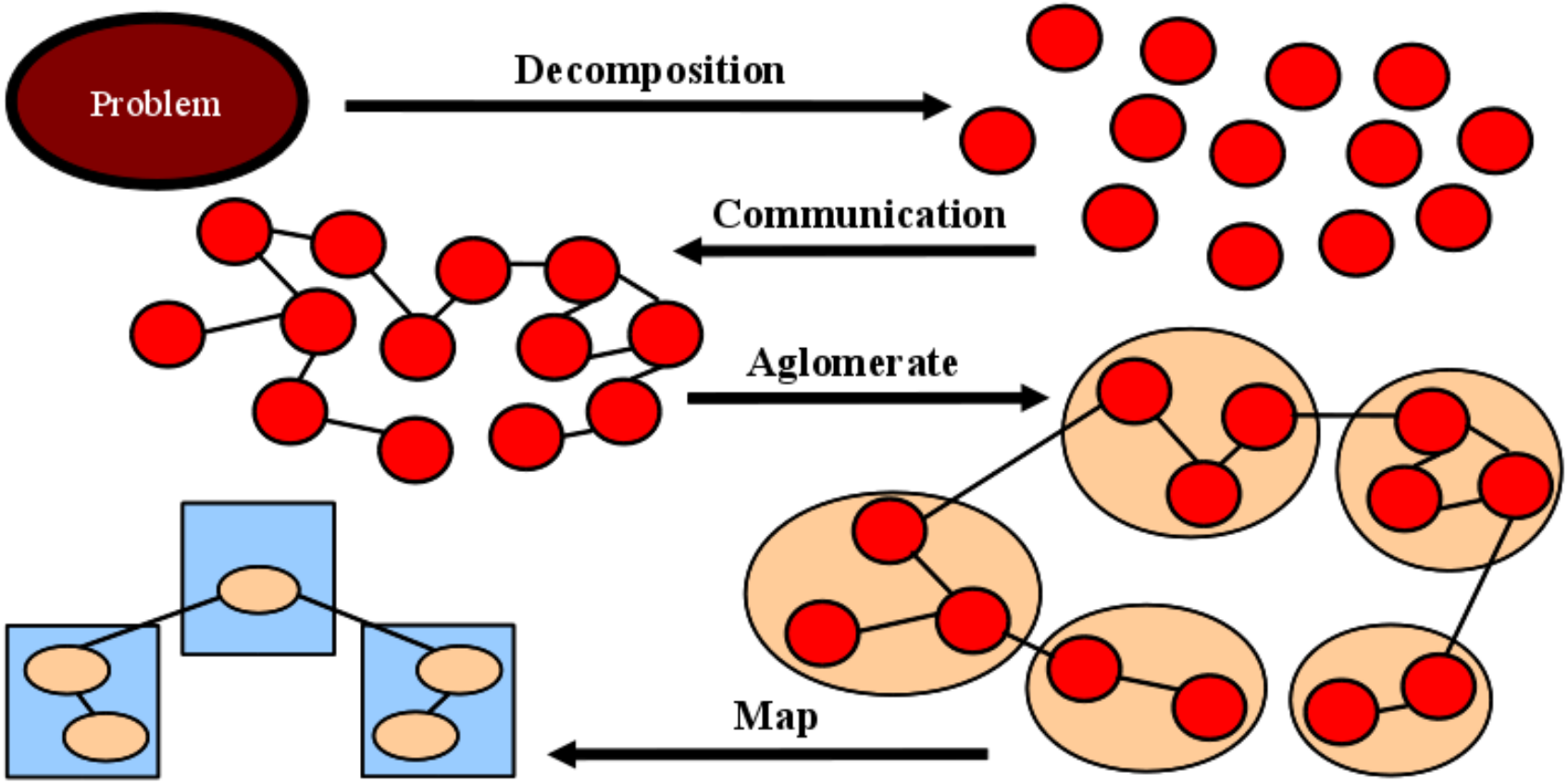
decomposing a program in smaller parts (or tasks) and efficiently mapping these tasks to available processors for parallel execution.

There are many difficulties:

- How to accomplish decomposition?
- How much should we divide?
- How to map efficiently tasks to processors?
- How to achieve cooperation and/or synchronization of non-independent tasks?
- How to gather results of tasks?
- ...

Designing Parallel Algorithms (Foster 96)

A metodologia de Foster envolve 4 etapas:

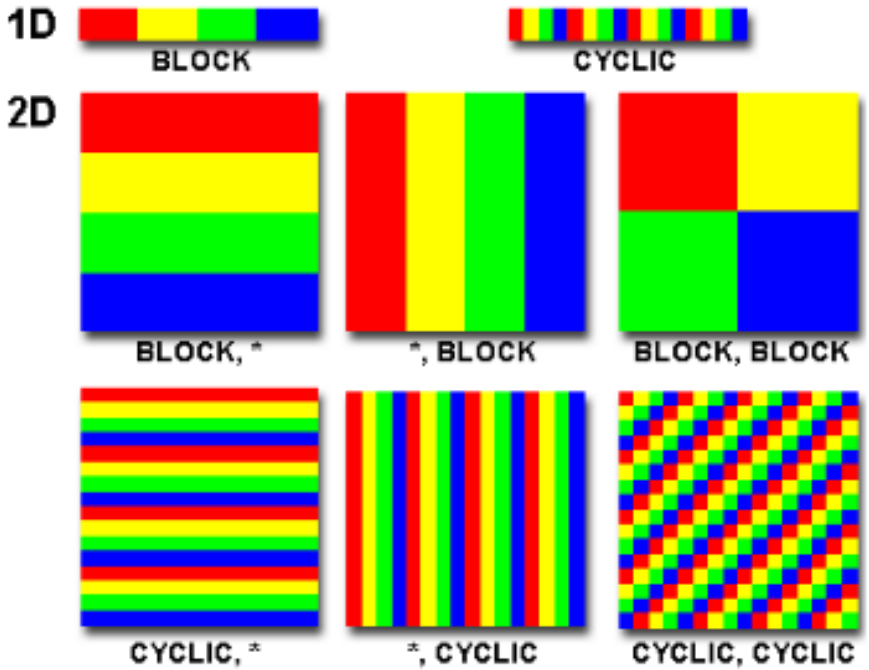


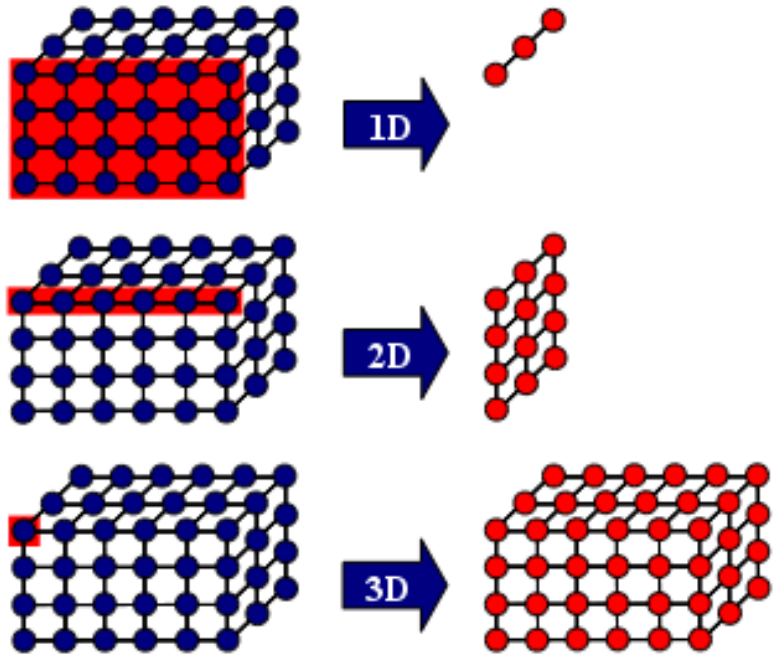
Decomposition

- Decomposing a problem into smaller problems, not only helps in reducing the complexity of the problem, but also allows for the sub-problems to be executed in parallel.
- There are two main strategies to decompose a problem:
  - Domain decomposition: decomposition based on the data.
  - Functional decomposition: decomposition based on the computation.
- A good scheme divides either data or computation, or both, into smaller tasks.

Domain Decomposition

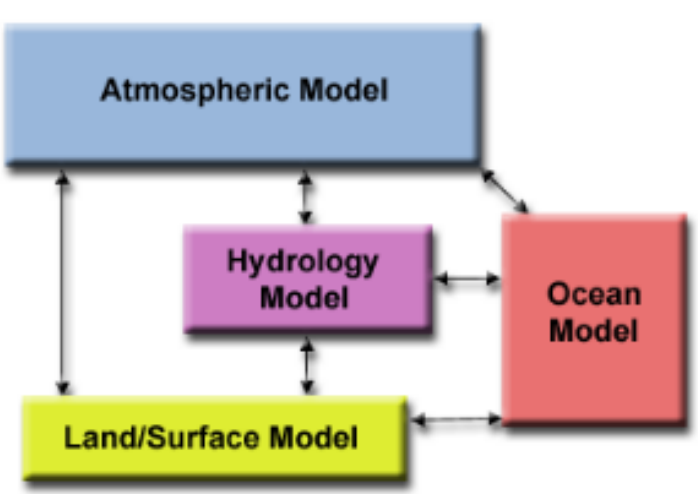
- First the data is partioned and only after we associate the computation to partitions.
- All tasks execute the same operations on different parts of data (data-parallelism).



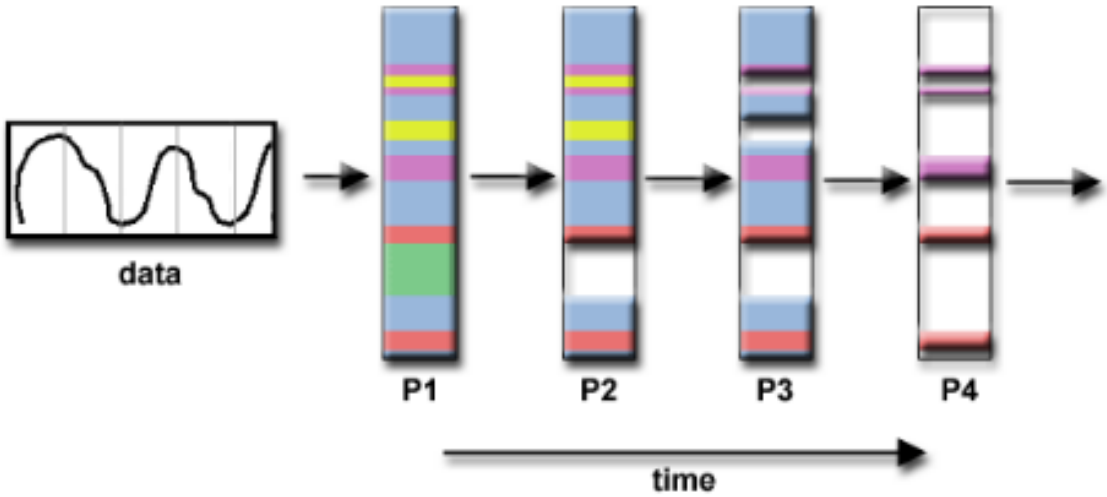


Functional Decomposition

- First we divide the computation in tasks and only after associate data with tasks.
- Different tasks may execute different operations on different data (functional parallelism).



Climate Modelling



Data Analysis

Communication

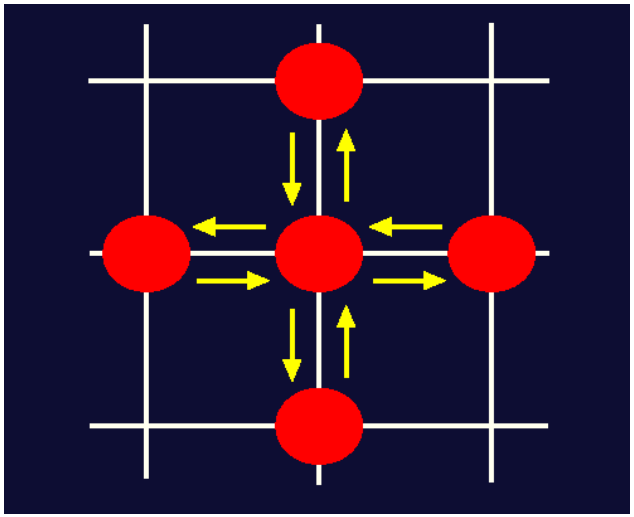
- The parallel execution of tasks might require:
  - **synchronization** as some tasks may only be executed after some other tasks have completed
  - **communication** between tasks to exchange data (e.g. partial results)
- Communication/synchronization can be a limiting factor for performance:
  - it has a **cost**. While you communicate, you do not compute!
  - **latency** - minimum time to communicate between two computing nodes
  - **bandwidth** - amount of data we can communicate per unit of time
- Avoid communicating too many small messages!

Communication Patterns (1)

- **Global vs. Local Communication**
  - tasks may communicate with any other task (Global), or
  - tasks just communicate with neighboring tasks (Local).

  Jacobi finite difference method:

$$X_{i,j}^{t+1} = \frac{4X_{i,j}^{t} + X_{i-1,j}^{t} + X_{i+1,j}^{t} + X_{i,j-1}^{t} + X_{i,j+1}^{t}}{8}$$

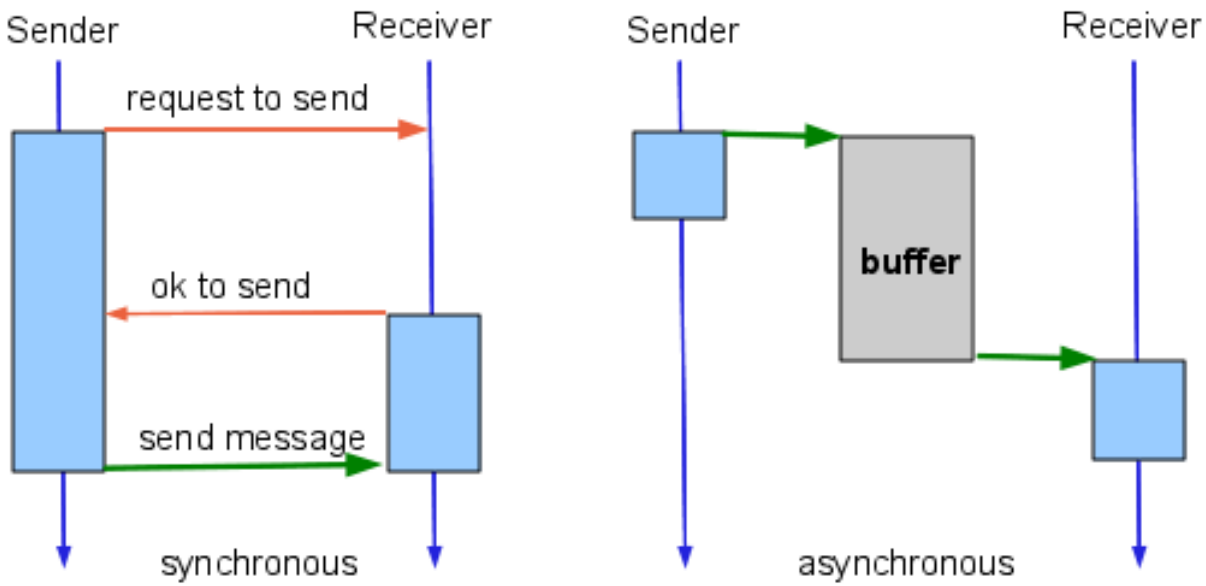


Communication Patterns (2)

- Structured Communication
  - communication between tasks follows a regular structure (e.g. a tree).
- Non-Structured Communication
  - communication between tasks follows an arbitrary graph
- Static Communication
  - communication pattern between tasks is known before execution.
- Dynamic Communication
  - communication between tasks is only determined during execution.

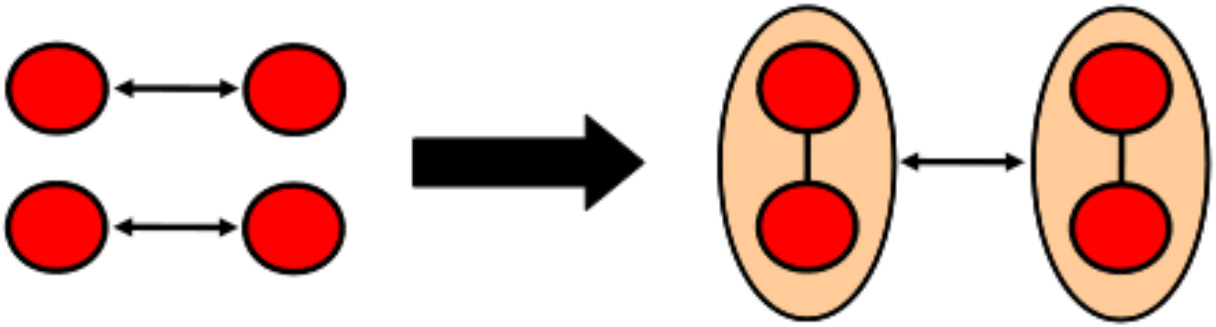
Communication Patterns (3)

- Synchronous communication
  - sender and receiver have to synchronize to start communicating (rendez-vous protocol)
- Asynchronous communication
  - sender writes messages to a buffer and continues execution; when ready, the receiver reads the messages from the buffer.
  - no agreement needed



Aggregation

- How small can the tasks be for parallel execution?
  - if too small, we incur in higher communication costs
  - time to compute the task must be higher then time to communicate it
- Granularity of tasks - measures the ratio between the average time to compute and to communicate the tasks
  - avoid too fine grained tasks!
- Aggregating small tasks into larger ones helps to reduce communication costs.



but, also do not over do it as with too large tasks we might limit available parallelism.

Granularity

- Granularity measures the size of a task.
  - it can be fine grain, medium grain, or coarse grain
  - The main question is? Which task size maximizes performance?
- Fine granularity:
  - smaller task sizes, less computation and more communication, thus smaller ratio between computation and communication.
  - (+) simplifies efficient workload balancing.
  - (-) computation cost of one task may not compensate for parallel costs (task creation, communication and synchronization).
- Coarse granularity:
  - larger ratio between computation and communication, thus, in principle, better efficiency.
  - (-) makes it difficult to achieve efficient workload balancing.
  - (+) computation costs of tasks compensate for parallel costs.

Mapping/Scheduling

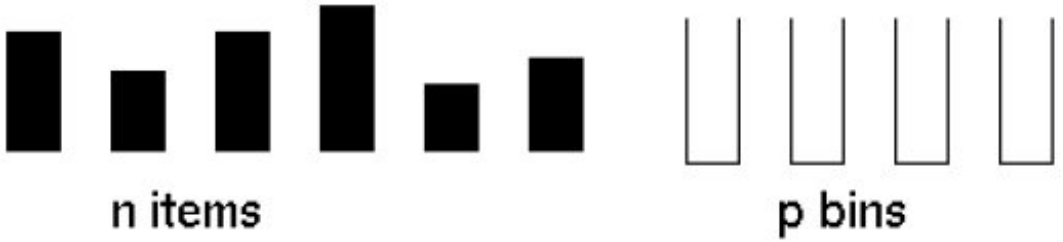
- To achieve maximum performance, one should:
  - maximize processor utilization (keep them busy computing the tasks), and
  - minimize communication/synchronization between processors.
- Thus, the question is how to best assign tasks to available processors to achieve maximum performance?
  - static scheduling - can be predetermined at compile time, normally with regular data-parallelism
  - dynamic scheduling - take decisions during execution and try to load balance work among the processors

Scheduling - cost of tasks

Cost of tasks (or granularity) influences decisions:

Easy: The tasks all have equal (unit) cost. branch–free loops

Harder: The tasks have different, but known, times. sparse matrix–vector multiply



Hardest: The task costs unknown until after execution. GCM, circuits, search

(slide from Kathy Yelick, CS267 lecture 24)

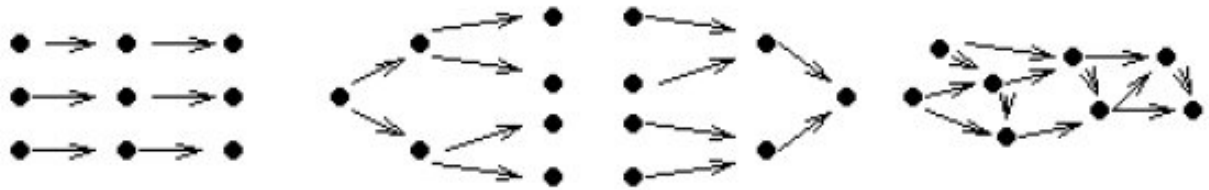
Scheduling - dependencies between tasks

Dependency between tasks influences decisions:

Easy: The tasks can execute in any order.

dependence free loops

Harder: The tasks have a predictable structure.



matrix computations (dense, and some sparse, Cholesky)

Hardest: The structure changes dynamically (slowly or quickly) search, sparse LU

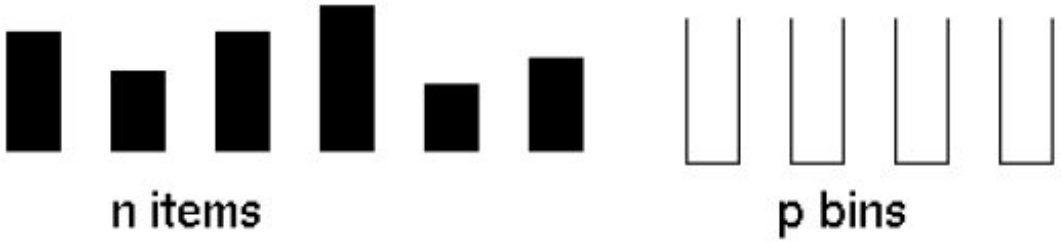
(slide from Kathy Yelick, CS267 lecture 24)

Scheduling - cost of tasks

Cost of tasks (or granularity) influences decisions:

Easy: The tasks all have equal (unit) cost. branch–free loops

Harder: The tasks have different, but known, times. sparse matrix–vector multiply



Hardest: The task costs unknown until after execution. GCM, circuits, search

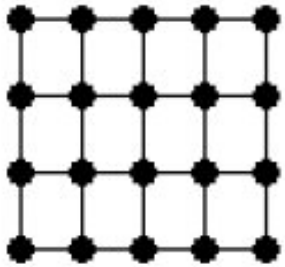
(slide from Kathy Yelick, CS267 lecture 24)

Scheduling - dependencies between tasks

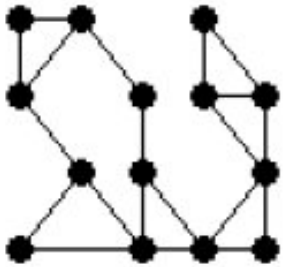
Dependency between tasks influences decisions:

Easy: The tasks, once created, do not communicate. — embarrassingly parallel

Harder: The tasks communicate in a predictable pattern.



regular

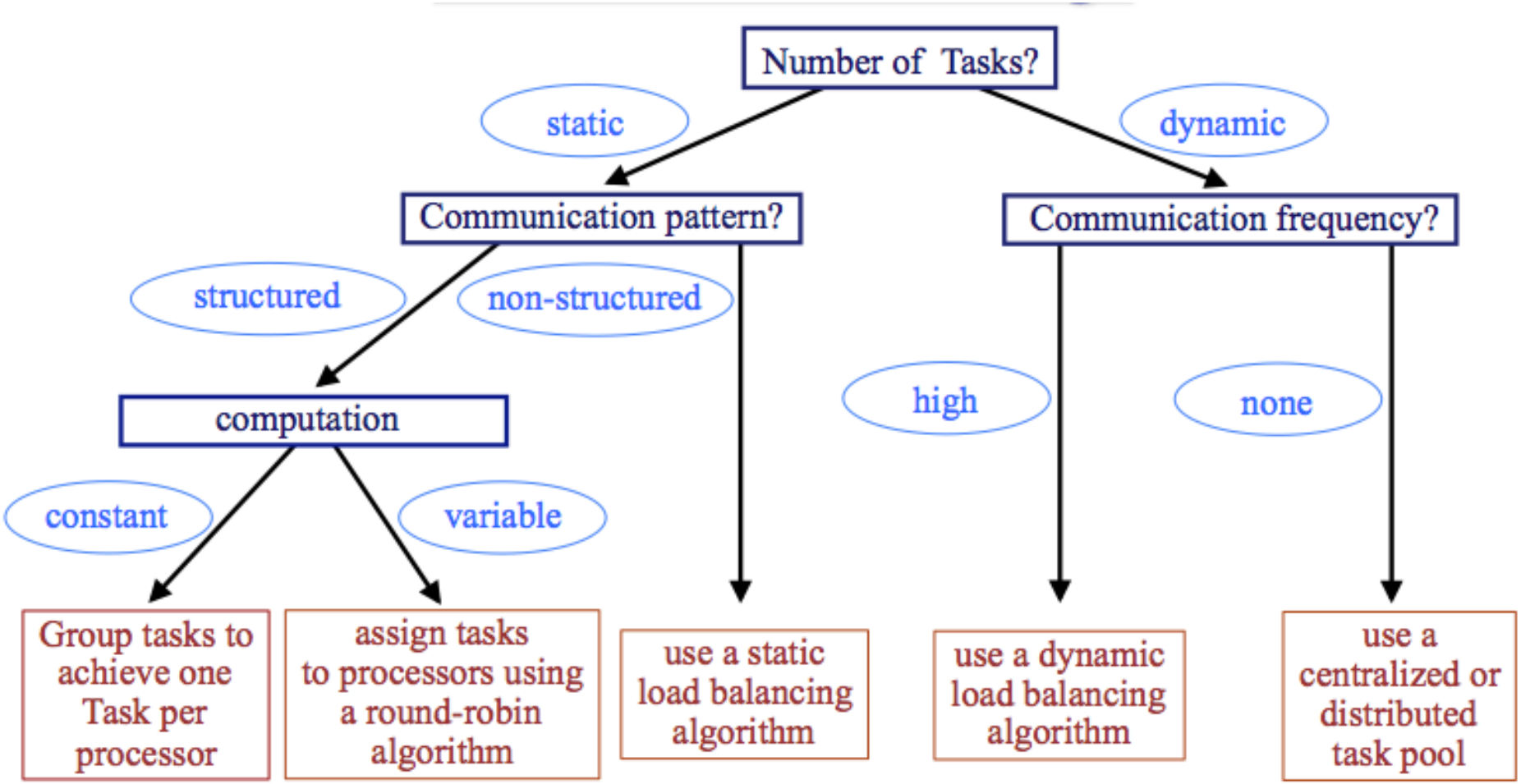


irregular

PDE solver

Hardest: The communication pattern is unpredictable. — discrete event simulation

Load Balancing / Workload Balancing

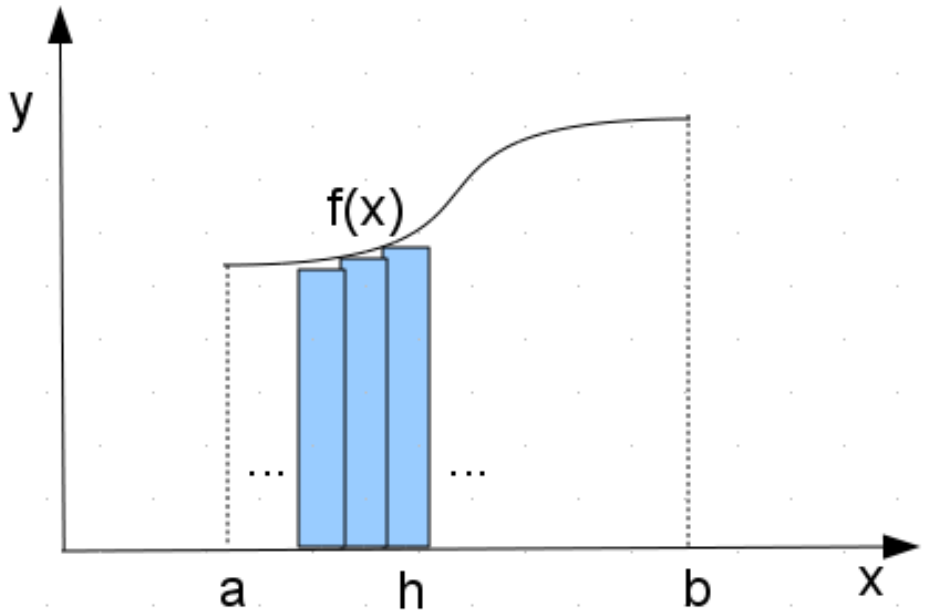


Main Parallel Programming Models

- Programming for Shared Memory
  - programming using processes, threads
  - communication via shared memory
  - synchronization using mutual exclusion mechanisms (e.g. locks)
  - environments: OpenMP (C/Fortran), Cilk++, C/C++ extensions
- Programming for Distributed Memory
  - preferable for large-grain tasks
  - communication and data sharing only via message exchange
  - environments: MPI (Message Passing Interface) most commonly used
- Hybrid Models
  - try to combine both models
  - mostly research environment tools

Example program - Numerical Integration

Consider calculating $\int_a^b f(x)dx$ using Numerical Integration by mid-point rule:



The idea is simple,

- divide interval $[a, b]$ in n intervals of size h, where $h = n/p$ and p is the number of processes.
- each process calculates the area in its interval.
- add partial results

Solution using MPI (1)

```
#include "mpi.h"
main()
{
int  myrank, p, n= 1024;
float a=0.0, b=1.0, h;                          all processors
float locA, locB, area, total;
int   locN, proc, dest=0, tag=0;
MPI_Status stat;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

h= (b-a)/n;
locN= n/p; /* # of rectangles in each process */

locA= a + myrank*locN*h; /* sub-interval for */
locB= locA + locN*h;        /* each process   */

/* calculations for all processes */
area= trap(locA, locB, locN, h);
```

Solution using MPI (2)

```
if (myrank==0) {
    total= area;                                          processor 0
    for (proc=1; proc<p; proc++) {
        MPI_Recv(&area,1,MPI_FLOAT,proc,tag,MPI_COMM_WORLD, stat);
        total= total + area;
    }
 }
 else                                                     processors 1..N-1
    MPI_Send(&area,1,MPI_FLOAT,dest,tag, MPI_COMM_WORLD);
 if (myrank==0)
    printf(``Area of Integral(%f,%f)= %f\n", a, b,total);   processor 0

 MPI_Finalize();                                          all processors
}
```

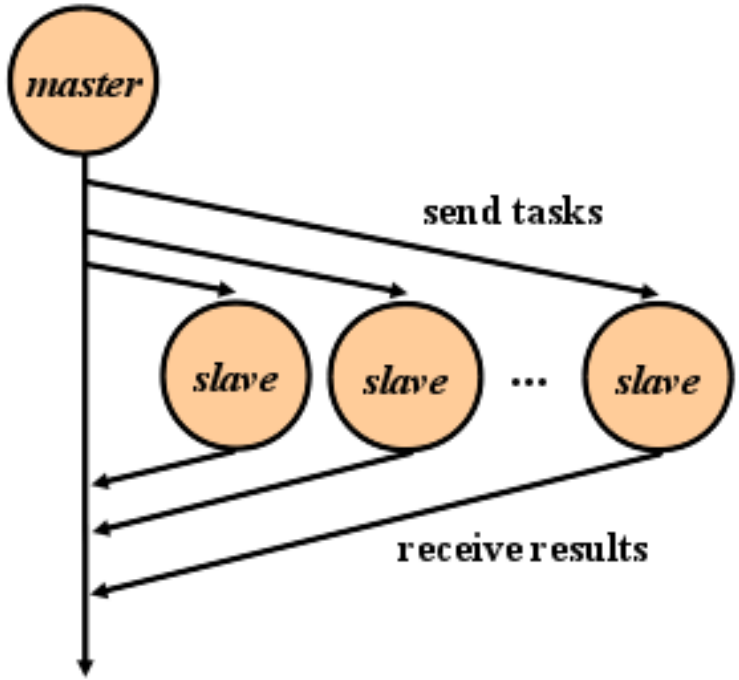
- What is happening?
  - all processors execute the same program
  - all processors initiate MPI and calculate their part of the result
  - processors 1 to N-1 send result to processor 0
  - processor 0 receives results from processors 1 to N-1
  - processor 0 prints the final result
  - all processors exit MPI

Main Parallel Programming Paradigms

- The following paradigms are the most commonly used:
  - Master/Slave
  - Single Program Multiple Data (SPMD)
  - Data Pipelining
  - Divide-and-Conquer or Tree-search
- Which paradigm should we use? It depends on the problem and
  - on the type of parallelism: data or functional parallelism
  - on the type of resources available, which might influence the granularity that can be exploited.

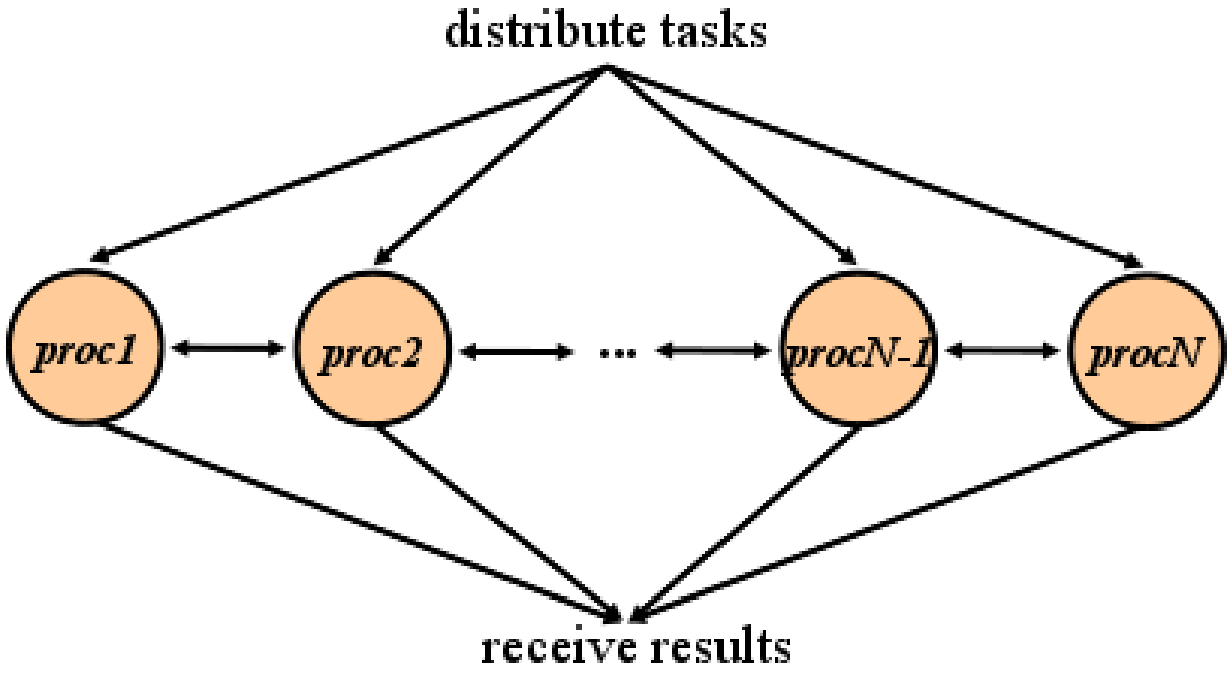
Master/Slave or Master/Worker

- A master process is responsible for
  - initiating the computation
  - possibly determine the tasks
  - distribute tasks to worker processors
  - aggregate partial results from workers, and produce final result
- slaves/workers execute a simpler execution cycle
  - receive task
  - compute task
  - send task-result to master



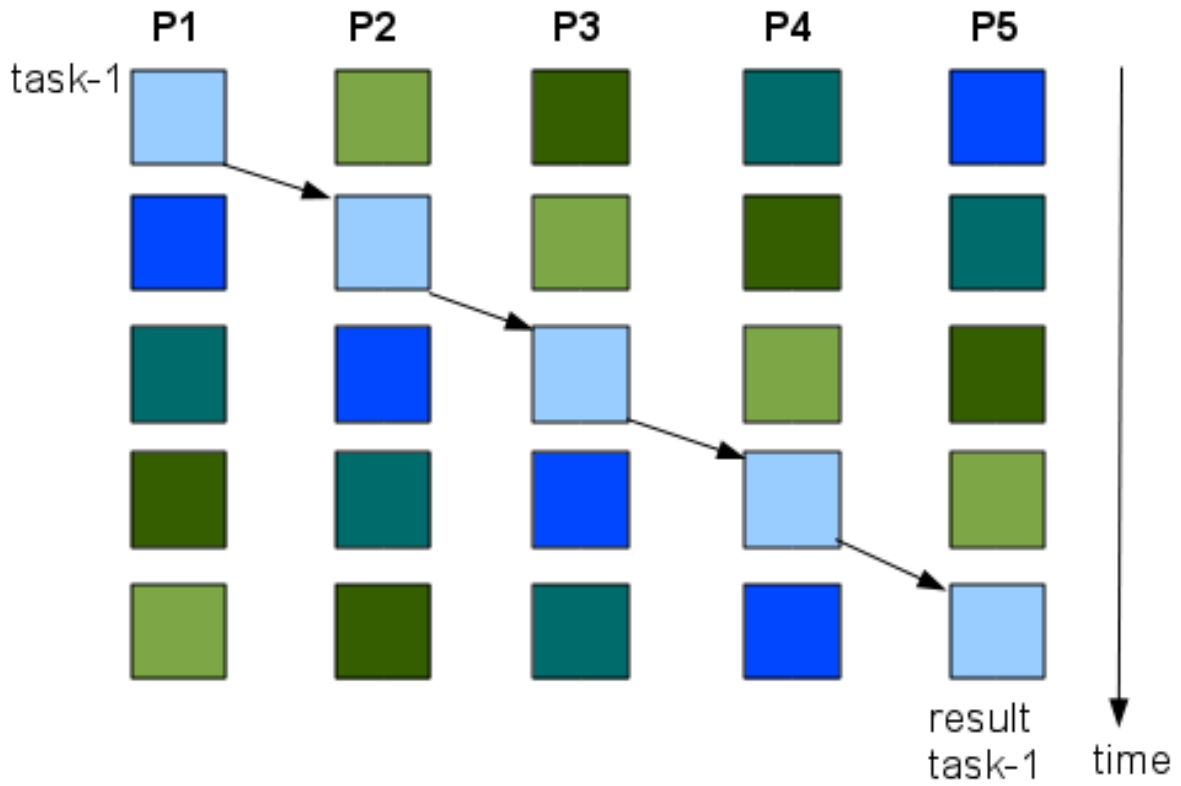
Single Program Multiple Data (SPMD)

- All processes execute the same program, but on different parts of data.
  - also known as data parallelism
  - similar to Master/Worker, but here we might have communication between tasks.



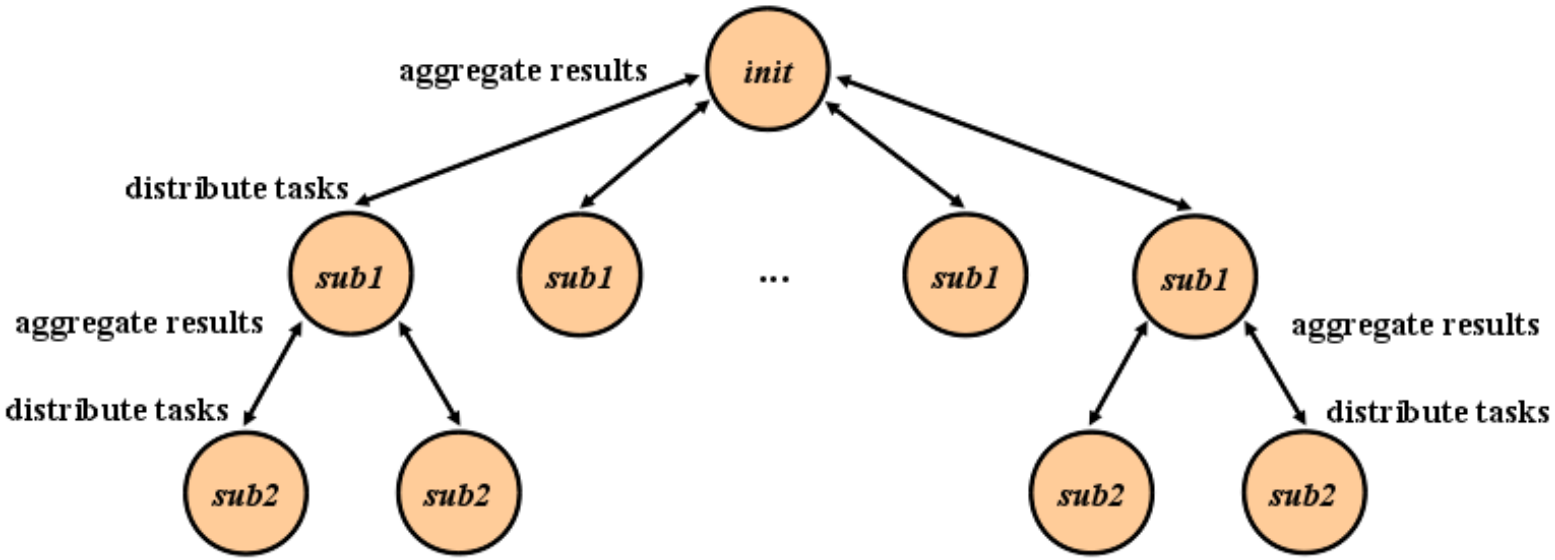
Data Pipelining

- Processes are organized in a pipeline fashion
  - for each task, each process does part of the computation
  - each process only communicates with the next process in the pipeline
- Parallelism is achieved by having multiple pipelines (i.e. tasks) being executed simultaneously



Tree-search or Divide-and-Conquer Parallelism

- For a large class of problems, their solution space corresponds to a search tree.
- It is the case for solutions that divide a problem in similar but less complex sub-problems, and solving these one can easily solve the initial problem.
  - these are normally computationally intensive
  - allow for a variety of parallelization strategies
  - require dynamic load balancing



Speculative Parallelism

- Used when dependencies are too complex that other pradigms are not sufficient.
- Parallelism is introduced by performing speculative computations
  - some related computations are anticipated, taking an optimistic assumption that they will be necessary.
  - if they are not necessary, they are terminated and some prior computation state may have to be recovered.
  - common in association with branch-and-bound algorithms

Programming Paradigms

The programming paradigms can also be differentiated by employing static or dynamic strategies for decomposition and/or mapping:

	Decomposition	Mapping
Master/Slave	static	dynamic
Single Program Multiple Data (SPMD)	static	static/dynamic
Data Pipelining	static	static
Divide and Conquer	dynamic	dynamic
Speculative Parallelism	dynamic	dynamic