Foundations

Ricardo Rocha and Fernando Silva

Computer Science Department
Faculty of Sciences
University of Porto

Parallel Computing 2017/2018

Why Go Parallel?

The scenario

If our best sequential algorithm can solve a given problem in N time units using 1 processing unit, could the same problem be solved in 1 time unit with a parallel algorithm using N processing units at the same time?

Why Go Parallel?

Major reasons to explore 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 memory limitations when the solution to some problems require more memory then one could find in just one computer
- Overcome the physical limitations in chip density and production costs of faster sequential computers

Simulation: the Third Pillar of Science

Traditional scientific and engineering paradigm:

- Do theory or paper design
- Perform experiments or build systems

Limitations of the traditional paradigm:

- Too difficult/expensive (e.g. build large wind tunnels)
- Too slow (e.g. wait for climate or galactic evolution)
- Too dangerous (e.g. weapons, drug design, climate experimentation)

Computational science paradigm:

- Based on known physical laws and efficient numerical methods, use high-performance computer systems to **simulate the phenomenon**

Grand Challenge Problems

Traditionally, the driving force for parallel computing has been the simulation of fundamental problems in science and engineering, with a strong scientific and economic impact, known as **Grand Challenge Problems (GCPs)**. Typically, GCPs simulate phenomena that cannot be measured by experimentation:

- Global climate modeling
- Earthquake and structural modeling
- Astrophysical modeling (e.g. planetary orbits)
- Financial and economic modeling (e.g. stock market)
- Computational biology (e.g. genomics, drug design)
- Computational chemistry (e.g. nuclear reactions)
- Computational fluid dynamics (e.g. airplane design)
- Computational electronics (e.g. hardware model checking)
- ...

New Data-Intensive Applications

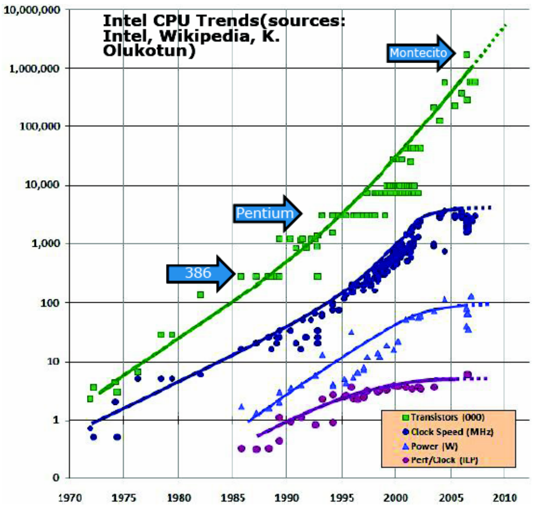
Currently, large volumes of data data are produced and their processing and analysis also require high performance computing:

- Data mining
- Web search
- Networked video
- Video games and virtual reality
- Computer aided medical diagnosis
- 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 and cooling problems



Free Lunch is Over (Herb Sutter, 2005)

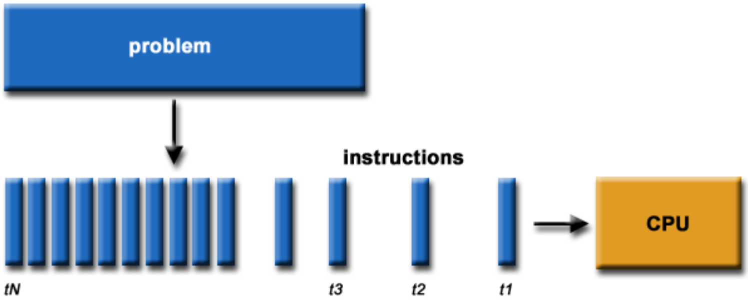
The manufacturer's solution was to start having **multiple cores** on the same chip and go for parallel computing.

This approach was not completely new, since chips already integrated many **Instruction-Level Parallelism (ILP)** techniques:

- Super pipelining
- Superscalar execution
- Out-of-order execution
- Branch prediction
- Speculative execution

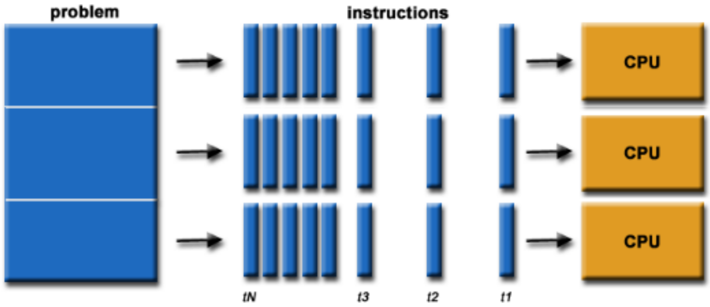
Sequential Computing

Sequential computing occurs when a problem is solved by executing **one flow of instructions in one processing unit**.



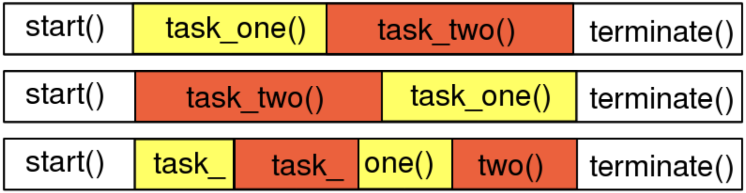
Parallel Computing

Parallel computing occurs when a problem is decomposed in multiple parts that can be solved **concurrently**. Each part is still solved by executing one flow of instructions in one processing unit but, as a whole, the problem can be solved by executing **multiple flows simultaneously using several processing units**.



Concurrency or Potential Parallelism

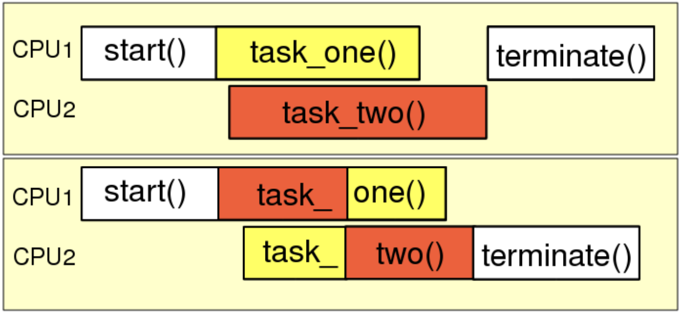
A program exhibits concurrency (or potential parallelism) when it includes **tasks (contiguous parts of the program) that can be executed in any order** without changing the expected result.



Parallelism

Parallelism is exploited when the concurrent tasks of a program are **executed simultaneously in more than one processing unit**:

- Smaller tasks simplify possible arrangements for execution
- Proportion of sequential tasks to start and terminate execution should be small as compared to the concurrent tasks



Implicit Parallelism

Parallelism is exploited implicitly when it is the **compiler** and the **runtime system** that:

- Automatically detect potential parallelism in the program
- Assign the tasks for parallel execution
- Control and synchronize execution

Advantages and disadvantages:

- **(+)** Frees the programmer from the details of parallel execution
- **(+)** More general and flexible solution
- **(−)** Very hard to achieve an efficient solution for specific problems

Explicit Parallelism

Parallelism is exploited explicitly when it is left to the **programmer** to:

- Annotate the tasks for parallel execution
- Assign tasks to the processing units
- Control the execution and the synchronization points

Advantages and disadvantages:

- **(+)** Experienced programmers achieve very efficient solutions for specific problems
- **(–)** Programmers are responsible for all details of execution
- **(–)** Programmers must have deep knowledge of the computer architecture to achieve maximum performance
- **(–)** Efficient solutions tend to be less/not portable between different computer architectures

Parallel Computational Resources

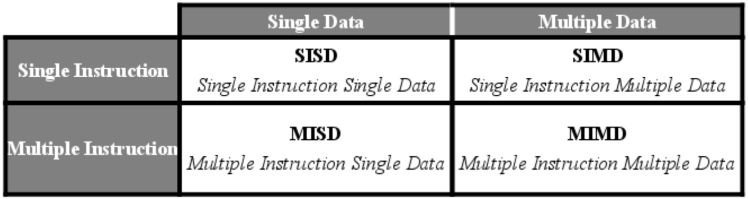
Putting it simply, we can define parallel computing as the use of multiple computational resources to reduce the execution time required to solve a given problem. Most common parallel resources include:

- **Multiprocessors (now also multicore processors)** – one machine with multiple processors/cores
- **Multicomputers** – an arbitrary number of dedicated interconnected machines
- **Clusters of multiprocessors and/or multicore processors** – a combination of the above

Flynn's Taxonomy (1966)

Flynn proposed a taxonomy to classify computer architectures that analyzes two independent dimensions available in the architecture:

- Number of **concurrent instructions**
- Number of **concurrent data streams**

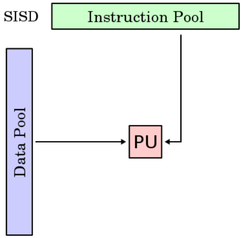


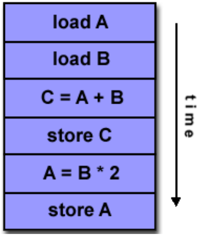
	Single Data	Multiple Data
Single Instruction	**SISD** *Single Instruction Single Data*	**SIMD** *Single Instruction Multiple Data*
Multiple Instruction	**MISD** *Multiple Instruction Single Data*	**MIMD** *Multiple Instruction Multiple Data*

SISD - Single Instruction Single Data

Corresponds to sequential architectures (no parallelism is possible):

- Only one instruction is processed at a time
- Only one data stream is processed at a time



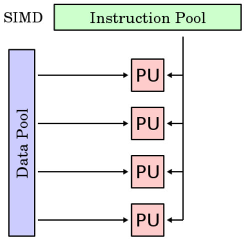


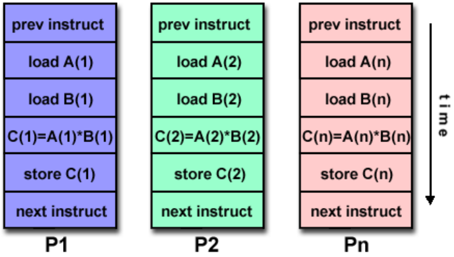
Examples: PCs, workstations and servers with one processor

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



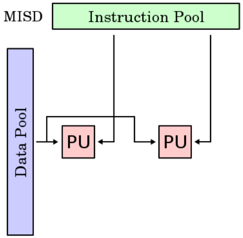


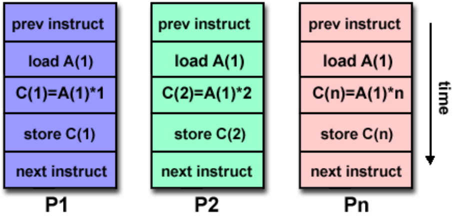
Examples: array processors and graphics processing units (GPUs)

MISD - Multiple Instruction Single Data

Uncommon parallel architecture where each processing unit performs a function on the same data stream (e.g. signal processing):

- 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



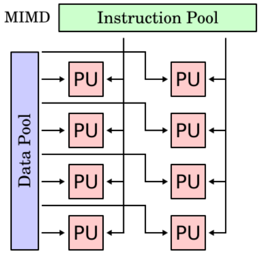


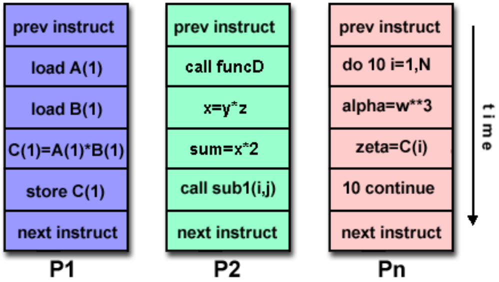
Examples: fault tolerance computers and systolic arrays

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



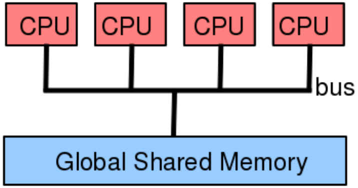


Examples: multiprocessors, multicore processors, multicomputers and clusters of multiprocessors and/or multicore processors

Multiprocessors

A multiprocessor or a shared memory machine is a 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

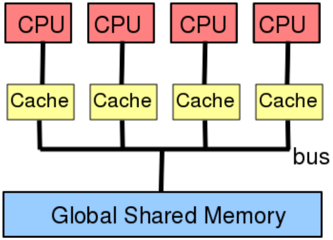
Classes of Multiprocessors

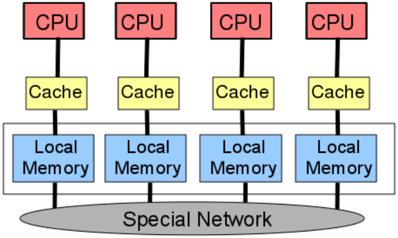
Uniform Memory Access (UMA)

- Equal access time to all memory
- Cache coherency implemented in hardware (write invalidate protocol)

Non-Uniform Memory Access (NUMA)

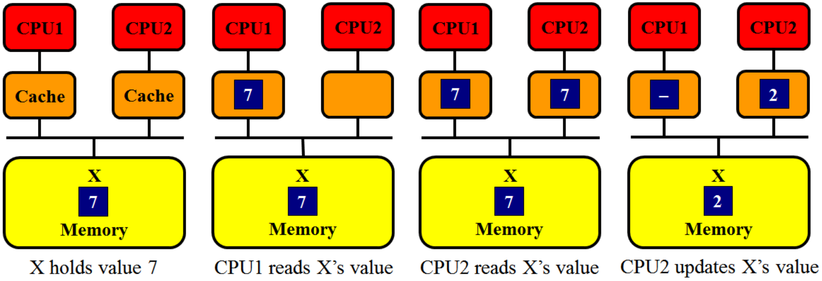
- Different access times to different memory regions
- Cache coherency implemented in hardware (directory-based protocol)





Write Invalidate Protocol

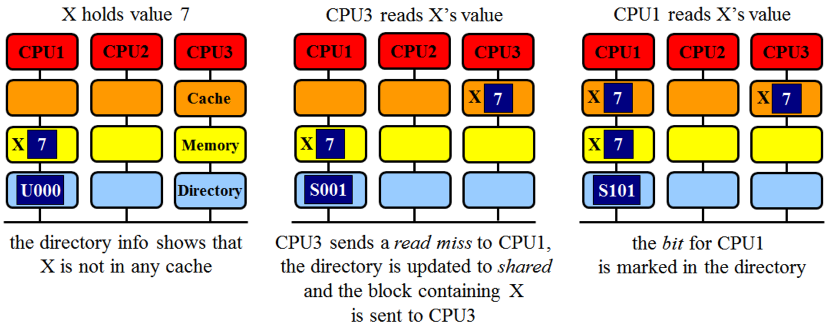
Before writing a value to memory, all existent copies in a processor cache are invalidated. Later, when a processor tries to access an invalidated value, a cache miss occurs and the value is reread from memory.



Directory-Based Protocol

A **directory data structure** holds state information about each processor's memory blocks. Blocks can be marked as:

- **Uncached** – not in any cache
- **Shared** – in one or more caches and the copy in memory is up-to-date
- **Exclusive** – only in one cache and the copy in memory is obsolete

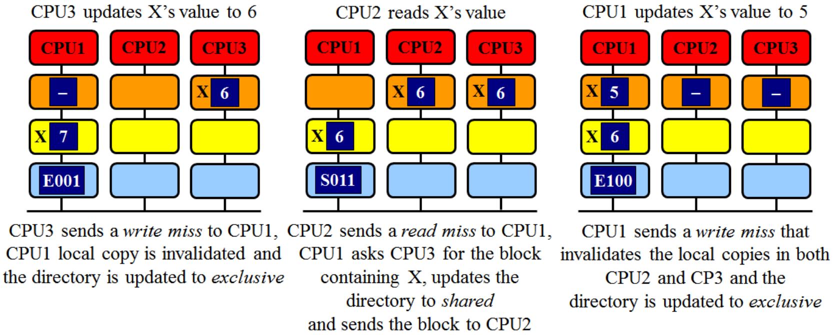


X holds value 7
CPU1
CPU2
CPU3
Cache
X 7
Memory
U000
Directory
the directory info shows that X is not in any cache
CPU3 reads X's value
CPU1
CPU2
CPU3
X 7
X 7
S001
CPU3 sends a *read miss* to CPU1, the directory is updated to *shared* and the block containing X is sent to CPU3
CPU1 reads X's value
CPU1
CPU2
CPU3
X 7
X 7
X 7
S101
the *bit* for CPU1 is marked in the directory

Directory-Based Protocol

A **directory data structure** holds state information about each processor's memory blocks. Blocks can be marked as:

- **Uncached** – not in any cache
- **Shared** – in one or more caches and the copy in memory is up-to-date
- **Exclusive** – only in one cache and the copy in memory is obsolete



CPU3 sends a *write miss* to CPU1, CPU1 local copy is invalidated and the directory is updated to *exclusive*

CPU2 sends a *read miss* to CPU1, CPU1 asks CPU3 for the block containing X, updates the directory to *shared* and sends the block to CPU2

CPU1 sends a *write miss* that invalidates the local copies in both CPU2 and CP3 and the directory is updated to *exclusive*

Multiprocessors

Advantages and disadvantages:

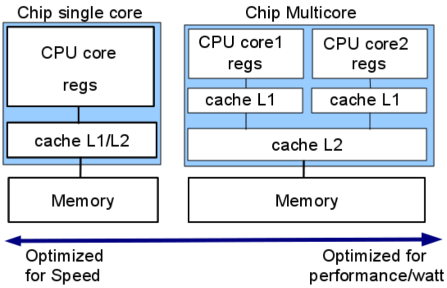
- **(+)** Simpler programming model as there is a global view of memory
- **(+)** Data sharing among concurrent tasks is simple, uniform and fast
- **(−)** Requires synchronization mechanisms to modify shared data
- **(−)** Not scalable, increasing the number of processors, increases bus congestion to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

Some of these disadvantages are being now overcome with new designs and by bringing the multiprocessor into the chip (the multicore processor).

Recent Multiprocessors - Multicore processors

Multicore **emphasizes shared memory parallelism**:

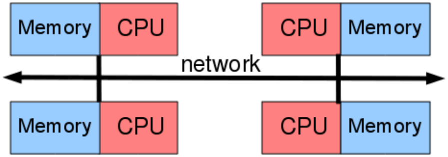
- Multicore processors are now the norm, reached mainstream desktops, game consoles, tablets and smartphones
- Supercomputers nowadays are clusters of multicore processors (number of cores exceeding 100,000 units)
- Leads to hybrid models of parallel programming



Multicomputers

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 and no global address space
- Each processor has its own address space
- Modifications on a memory position by a processor are not visible by other processors
- Data sharing or synchronization takes place by exchanging messages

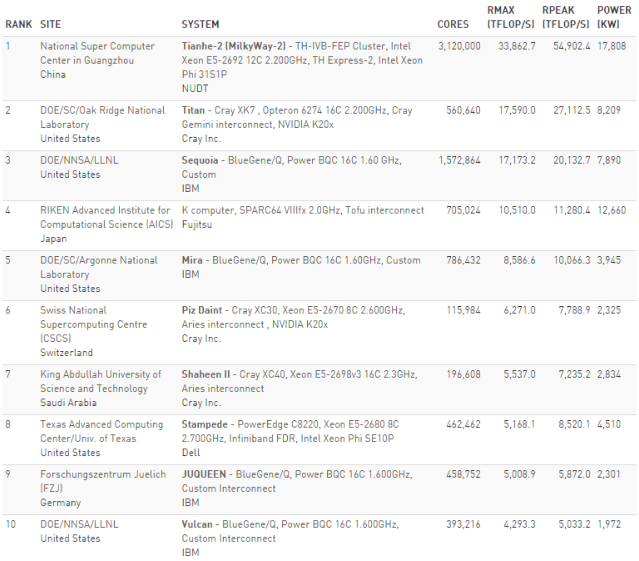


Multicomputers

Advantages and disadvantages:

- **(+)** High scalability on processors and memory (no cache coherency mechanisms required)
- **(+)** Reduced cost, in fact they can be built using off-the-shelf components (Beowulf cluster)
- **(–)** Communication and synchronization via message exchange only
- **(–)** Remote data access is very costly in performance
- **(–)** Harder to program, as the programmer has to control explicitly communication. Moreover, it can be difficult to convert/adapt data structures for shared memory to be used in distributed memory

Top500 Supercomputers List (www.top500.org)



RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	King Abdullah University of Science and Technology Saudi Arabia	**Shaheen II** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	196,608	5,537.0	7,235.2	2,834
8	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
9	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301
10	DOE/NNSA/LLNL United States	**Vulcan** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393,216	4,293.3	5,033.2	1,972

List for June 2015 (list updated twice a year)

Parallel Programming

We looked at different types of parallel architectures, but the main question is how can we develop software that takes advantage of their full computing capacity?

There are many difficulties that do not exist in sequential programming:

- **Concurrency** – which parts of the computation (tasks) can be executed concurrently?
- **Communication and synchronization** – how to achieve cooperation and/or synchronization of non-independent tasks and how to gather results of tasks?
- **Load balancing and scheduling** – how much should we divide and how to map efficiently tasks to processors/cores?

Parallel Programming

The truth

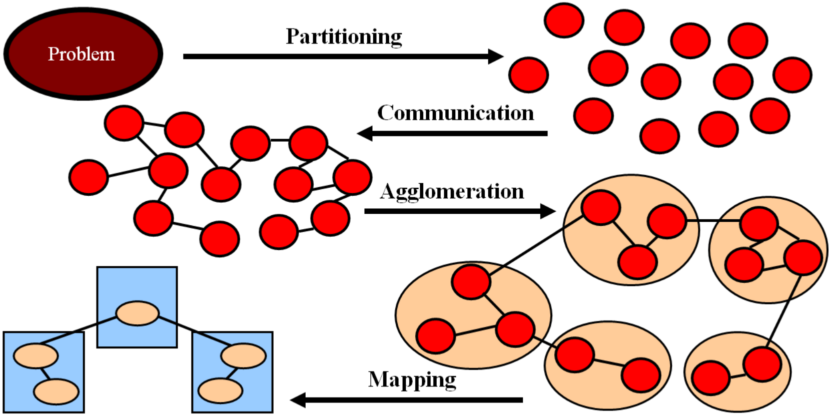
Parallel programming remains a very complex task!

Foster's Design Methodology

It is not easy to design a parallel algorithm from scratch without some logical methodology. It is far better to use a proven methodology that is general enough and that can be followed easily. In 1995, Ian Foster proposed such a methodology, which has come to be called Foster's design methodology. Foster's methodology involves 4 steps:

- **Partitioning** – the process of dividing the computation and the data into pieces
- **Communication** – the process of determining how tasks will communicate with each other
- **Agglomeration** – the process of grouping tasks into larger tasks to improve performance or simplify programming
- **Mapping** – the process of assigning tasks to physical processors

Foster's Design Methodology



Decomposition

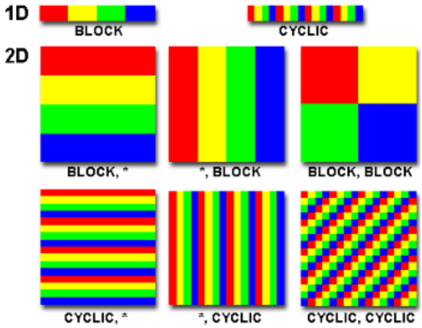
Decomposing a problem into smaller problems, not only helps in reducing the complexity of the problem, but also allows for the sub-problems to be executed in parallel. There are two main strategies to decompose a problem:

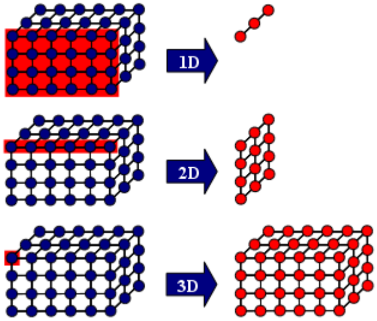
- **Domain decomposition** – decomposition based on the data
- **Functional decomposition** – decomposition based on the computation

A good decomposition scheme divides both data and computation into smaller tasks.

Domain Decomposition

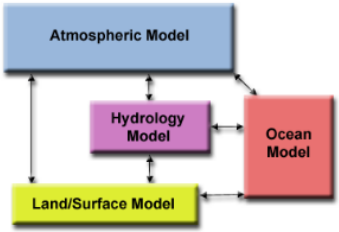
First the data is partitioned and only after we associate the computation to partitions. All tasks execute the same operations on different parts of data (data parallelism).



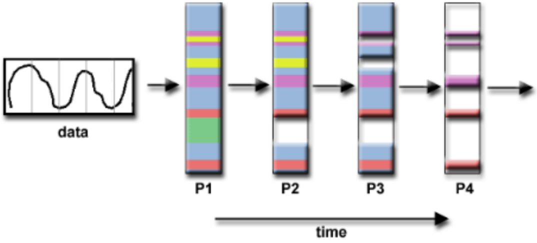


Functional Decomposition

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:

- **Communication** between tasks to exchange data (e.g. partial results)
- **Synchronization** as some tasks may only be executed after some other tasks have completed

Communication/synchronization can be a limiting factor for performance:

- **Implicit cost** – while you communicate/synchronize, you do not compute!
- **Latency** – minimum time to communicate between two computing nodes
- **Bandwidth** – amount of data we can communicate per unit of time

Good practice: avoid communicating too many small messages!

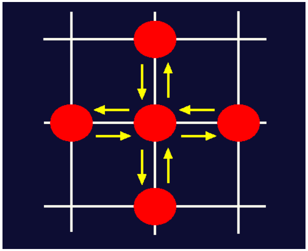
Communication Patterns

Global communication:

- Tasks may communicate with any other task

Local communication:

- Tasks just communicate with neighboring tasks (e.g. 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

Structured communication:

- Communication between tasks follows a regular structure (e.g. 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 known during execution

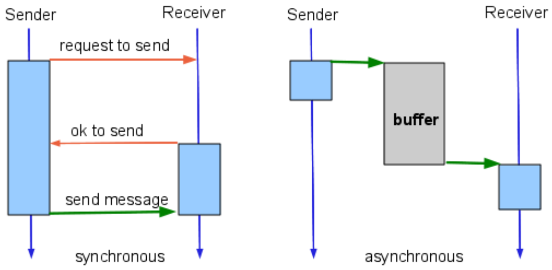
Communication Patterns

Synchronous communication:

- Sender and receiver have to synchronize to start communicating (e.g. rendez-vous protocol)

Asynchronous communication:

- No agreement needed, sender writes messages to a buffer and continues execution
- When ready, the receiver reads the messages from the buffer

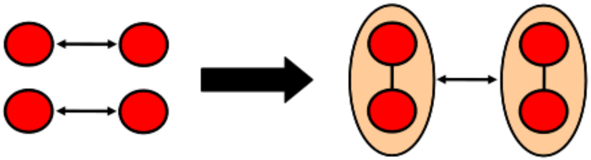


Agglomeration

How small can tasks be for parallel execution?

- Time to compute a task must be higher than the time to communicate it
- Smaller tasks, leads to more communication between them

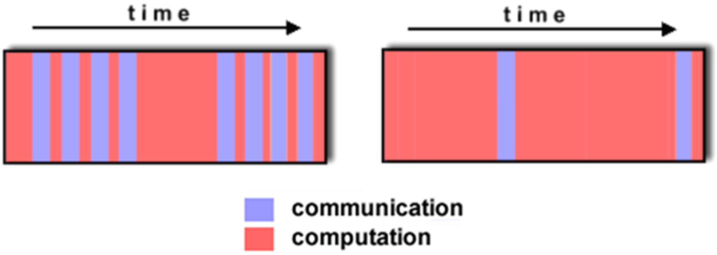
Aggregating small tasks into larger ones might help to reduce communication costs but, by over doing it (i.e., with too large tasks), we might be limiting the available parallelism.



Granularity of Tasks

Granularity measures the ratio between the time doing computation and the time doing communication

- It can be fine grain, medium grain, or coarse grain
- The main question is **which task size maximizes performance?**



Granularity of Tasks

Fine granularity:

- Computation grouped as a big number of small tasks
- Low ratio between computation and communication
- **(+)** Simplifies efficient workload balancing
- **(−)** Computation cost of one task may not compensate the parallel costs (task creation, communication and synchronization costs)
- **(−)** Difficult to improve performance

Coarse granularity:

- Computation grouped as a small number of big tasks
- High ratio between computation and communication
- **(−)** Difficult to achieve efficient workload balancing
- **(+)** Computation costs compensate the parallel costs
- **(+)** More opportunities to improve performance

Mapping

To achieve maximum performance, one should:

- Maximize processor occupation (keep them busy computing tasks)
- Minimize communication/synchronization between processors

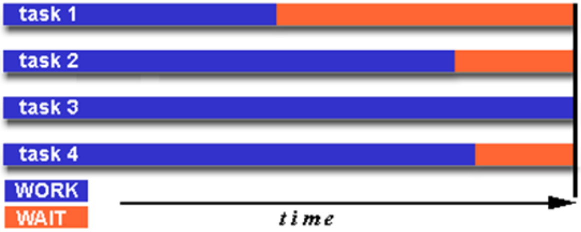
Thus, the question is **how to best assign tasks to available processors to achieve maximum performance?**

- The percentage of occupation is optimal when the computation is equally divided by the available processors, allowing them to start and finish their tasks simultaneously
- The percentage of occupation decreases when one or more processors are idle while the others stay busy

Load Balancing

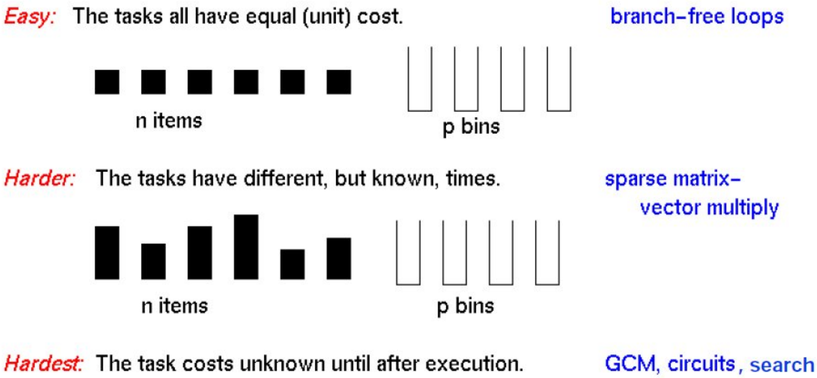
Load balancing can be seen as a scheduling procedure that tries to minimize the time processors are not busy:

- **Static scheduling** – can be predetermined at compile time, normally with regular data parallelism
- **Dynamic scheduling** – decisions are taken during execution trying to load balance work among the available processors



Scheduling Decisions

Granularity of tasks influences decisions:



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

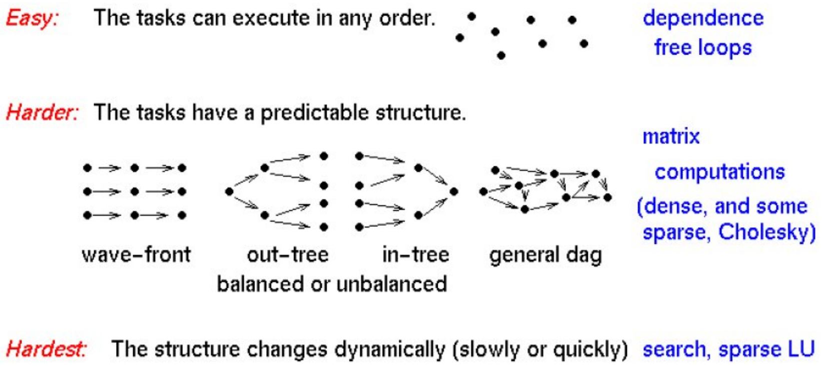
Harder: The tasks have different, but known, times. sparse matrix–vector multiply

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

(figure from Kathy Yelick, CS267 lecture 24)

Scheduling Decisions

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

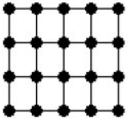
(figure from Kathy Yelick, CS267 lecture 24)

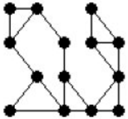
Scheduling Decisions

Dependency between tasks influences decisions:

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

Harder: The tasks communicate in a predictable pattern. PDE solver

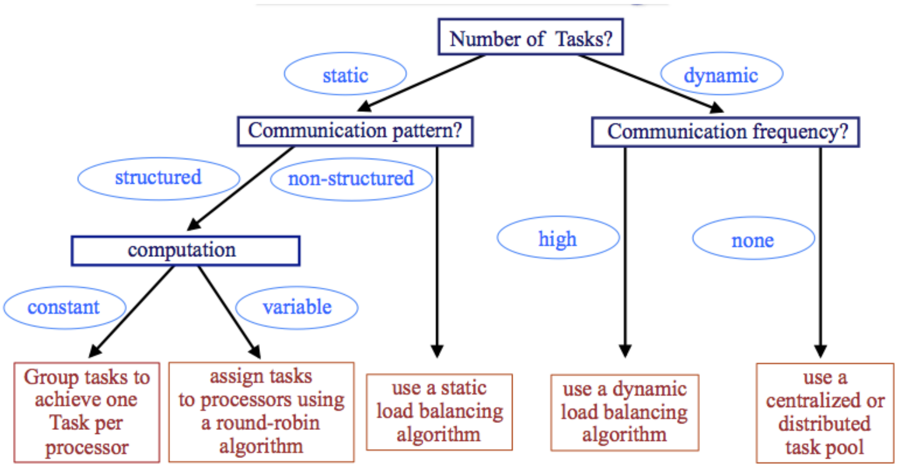

regular


irregular

Hardest: The communication pattern is unpredictable. discrete event simulation

(figure from Kathy Yelick, CS267 lecture 24)

Load Balancing Decision Tree


Number of Tasks?
static
dynamic
Communication pattern?
Communication frequency?
structured
non-structured
high
none
computation
constant
variable
Group tasks to achieve one Task per processor
assign tasks to processors using a round-robin algorithm
use a static load balancing algorithm
use a dynamic load balancing algorithm
use a centralized or distributed task pool

Main Parallel Programming Models

Programming for shared memory

- Programming using processes and/or threads
- Communication via shared memory
- Synchronization using mutual exclusion mechanisms (e.g. locks)
- Environments and tools: shared memory segments, Pthreads and OpenMP

Programming for distributed memory

- Preferable for large-grain tasks
- Communication and data sharing only via message exchange
- Environments and tools: MPI

Hybrid programming models

- Try to combine both models
- Environments and tools: MPI/Threads and MPI/OpenMP

Main Parallel Programming Paradigms

Despite the diversity of problems where parallel programming can be applied, the kind of paradigms used to solve such problems can be classified in a very small set of different approaches. The following paradigms are the most commonly used:

- **Master/Slave**
- **Single Program Multiple Data (SPMD)**
- **Data pipelining**
- **Divide-and-conquer**
- **Speculative parallelism**

Which paradigm should we use depends on the:

- Type of parallelism, domain or functional
- Type of available resources, which might influence the granularity that can be exploited

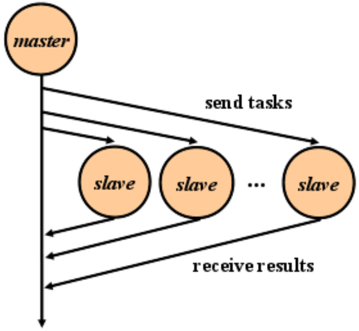
Master/Slave

A **master** process is responsible for:

- Decompose the problem into tasks
- Distribute the tasks to the slaves
- Aggregate partial results and produce the final result

The set of **slaves** follow a simpler execution cycle:

- Receive a task from the master
- Compute the task
- Send the task result back to the master



Master/Slave

Load balancing can be static or dynamic:

- If static, the master can also participate in the computation
- If dynamic, the slaves ask for new tasks when they have finished the current one

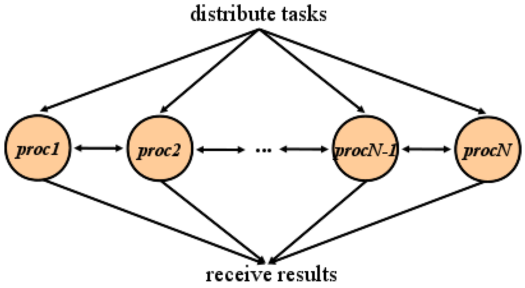
Advantages and disadvantages:

- **(+)** Reduced communication, each slave only communicates with the master and a few number of times
- **(–)** Centralized control can be a problem when we increase the number of slaves (use several masters instead, each one controlling a different set of slaves)

Single Program Multiple Data (SPMD)

All processes execute the same program (binary), but on different parts of data (also known as **data parallelism**)

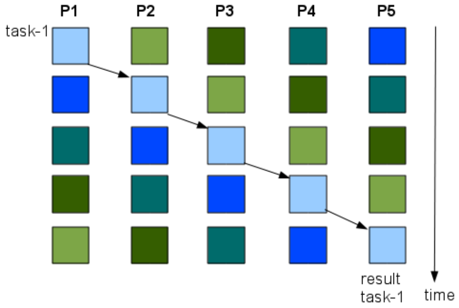
- Similar to Master/Slave, but here we might have communication between tasks
- Typically, the tasks have equal cost and the communication pattern is mostly local, structured and static, which allows for good performance and scalability



Data Pipelining

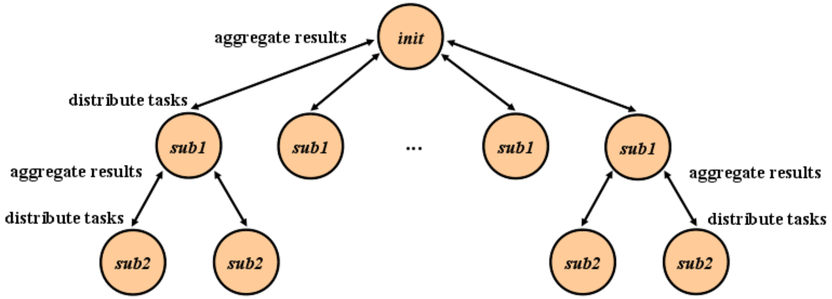
Follows a functional decomposition of the problem where processes are organized in a pipeline fashion (also known as **data flow parallelism**):

- For each task, each process does a part of the computation
- Each process only communicates with the next process in the pipeline
- Parallelism is achieved by having multiple pipelines being executed simultaneously



Divide-and-Conquer

Works by recursively breaking down a problem into sub-problems of the same type until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.



Divide-and-Conquer

The computation can be seen like a virtual tree:

- The leaf nodes compute the sub-tasks
- The remaining nodes are responsible to create the sub-tasks and to aggregate the partial results

Advantages and disadvantages:

- **(+)** Reduced communication, each node only communicates with its children to distribute tasks and aggregate results
- **(+)** Allows for a variety of parallelization strategies
- **(−)** Requires dynamic load balancing to distribute sub-tasks among processes

Speculative Parallelism

Used when data dependencies are too complex and do not fit within the other paradigms. Parallelism is introduced by performing speculative and/or out-of-order computations:

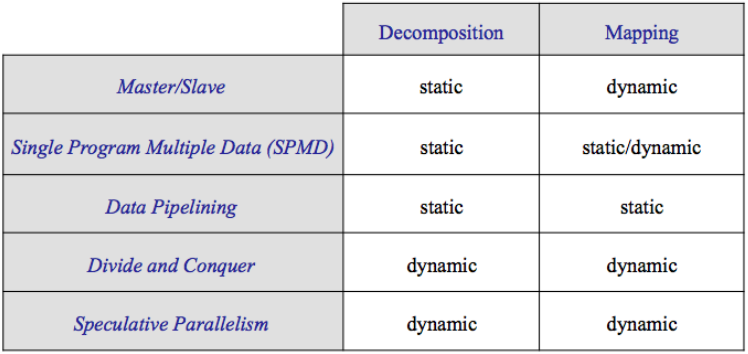
- Some related computations are anticipated, taking an optimistic assumption that they will be necessary
- Later, if they are not necessary, they are terminated and some prior computation state may have to be recovered

Also common in association with branch-and-bound algorithms:

- A set of candidate sub-tasks is set off to be explored in parallel
- The first or better solution found is used to prune the search space for the set of candidate sub-tasks

Programming Paradigms

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