Computação de Alto Desempenho (High Performance Computing)

MIEI 2015/16

Overview

- Course description
- Workload and grading
- Bibliography
- Course goals

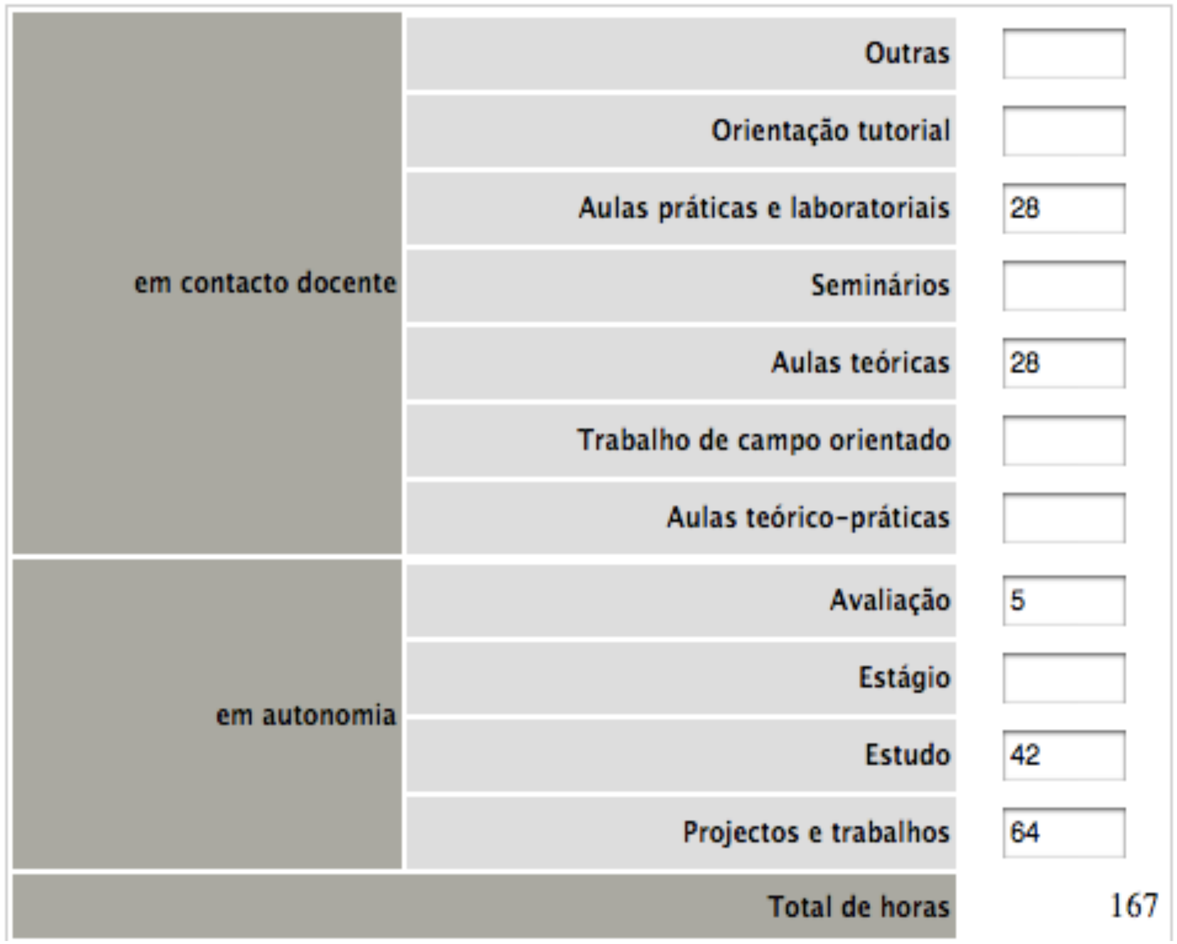
Course administrative details

- 6 credits (ECTS)
- 1 credit = 28 hours of work
- Contact hours
  - Lectures (2h weekly)
  - Labs (2h + 1 weekly)
- Self-study
  - Preparation of tests, exams and programming assignments

Workload

Contact hours →

Self study →



em contacto docente	Outras	
	Orientação tutorial	
	Aulas práticas e laboratoriais	28
	Seminários	
	Aulas teóricas	28
	Trabalho de campo orientado	
	Aulas teórico-práticas	
em autonomia	Avaliação	5
	Estágio	
	Estudo	42
	Projectos e trabalhos	64
Total de horas		167

167 hours = 11 hours / week = 7 h outside the classes

Prerequisites

- 1st cycle level courses
  - Computer architecture
  - Operating systems
  - Computer networks
- Programming languages
  - C / C++ (, Java)

Grading

- Tests/Exam grade TG
  - Two tests (NT1 and NT2), or
  - Written exame NR (only “Recurso”)
  - Grade TG = 0.5*NT1+0.5*NT2 or TG = NR
  - Minimum grade: 8.5 points (out of 20)
- Labs grade LG
  - 3 programming assignments, delivered with source code and a small report
  - LG = (NP1+NP2+NP3)/3
- Final grade 0.6*TG+0.4*LG

Evaluation dates (to be confirmed)

- Tests
  - Week of 2016-04-12
  - Week of 2016-06-7
- Assignments/Labs
  - TP1: 2016-04-10, Sunday
  - TP2: 2016-05-8, Sunday
  - TP3: 2016-05-26, Thursday

Bibliography

- Main:

  P. Pacheco, " An Introduction to Parallel Programming", Morgan Kauffman, 2011

- Complementary:
  - Norm Matloff, "Programming on Parallel Machines: GPU, Multicore, Clusters and More", University of California, Davis, http://heather.cs.udavis.edu/~matloff/158/PLN/ParBook.pdf
  - Bibliography on CUDA

- CLIP (*activate the notifications option*)

CAD/HPC goal

- The main goal is to give competences in the area of High Performance Computing; i.e. to the methodologies and techniques that allow the exploitation of hardware architectures with multiple processors, in order to reduce the execution times of programs that need high computational resources.

Topics

1. Parallel computing: hardware, software, applications and performance theory. (2 weeks)
2. Programming shared-memory multiprocessors. OpenMP. Application examples. (3 weeks)
3. GPU computing. CUDA. (3 weeks)
4. Programming distributed-memory multiprocessors. MPI. Application examples. (3 weeks)
5. Structured Parallel Programming -- structured models and abstractions for parallel programming. (1 week)

Parallel Computing

Roadmap

- Why we need ever-increasing performance.

Parallelism is a familiar concept

- House construction – several workers can perform separate tasks simultaneously
- Manufacturing (e.g. cars) – it is performed in parallel using an assembly line, or pipeline → many units of the product under construction at the same time
- Call center – many employees service customers at the same time
- Etc.

Need for increasing performance

- Complex problems, e.g.
  - Climate modeling
  - Drug discovery
  - Energy Research
  - Intelligent Transport Systems
  - The Big Data problem

The Big Data problem

Data management dimensions:

- Data acquisition (e.g. sensors everywhere)
- Data archiving / mining (e.g. databases, search engines)
- Data processing (online/streaming and offline data)
- Data access / dissemination (e.g. large-scale applications with high number of users)

Roadmap

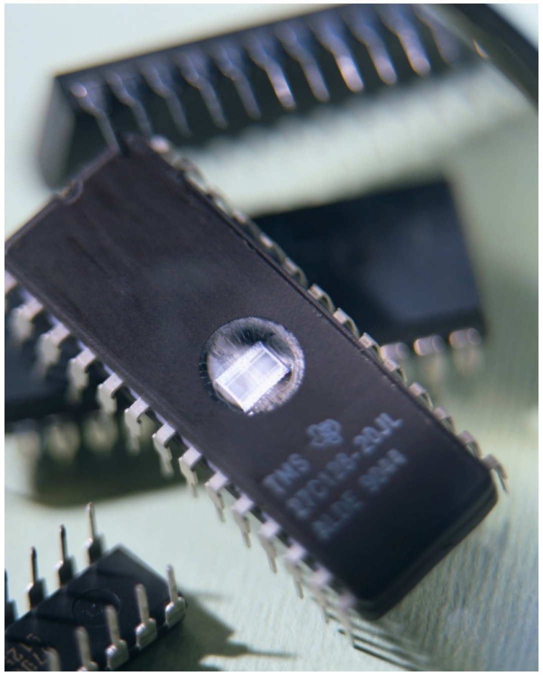
- Why we need ever-increasing performance.
  - More complex problems
- Why we are building parallel systems.

  Up to now, performance increases have been attributable to increasing density of transistors.

  However, there are inherent problems:
    - Smaller transistors = faster processors.
    - Faster processors = increased power consumption.
    - Increased power consumption = increased heat.
    - Increased heat = unreliable processors.

Solution

- Move away from single-core systems to multicore processors.
- “core” = central processing unit (CPU)



- Introducing parallelism!!!

Roadmap

- Why we need ever-increasing performance.
  - More complex problems
- Why we are building parallel systems.
- Why we need to write parallel programs.

Why we need to write parallel programs

- Running multiple instances of a serial program often is not very useful.
- Think of running multiple instances of your favorite game.
- What you really want is for it to run faster.

Approaches to the serial problem

- Rewrite serial programs so that they are parallel.

- Write translation programs that automatically convert serial programs into parallel programs.
  - This is very difficult to do.
  - Success has been limited.

More problems

- Some coding constructs can be recognized by an automatic program generator, and converted to a parallel construct.
- However, it is likely that the result will be a very inefficient program.
- Sometimes the best parallel solution is to step back and devise an entirely new algorithm.

Example

- Compute n values and add them together.
- Serial solution:

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Example (cont.)

- We have p cores, p much smaller than n.
- Each core performs a partial sum of approximately n/p values.

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
    my_x = Compute_next_value( . . .);
    my_sum += my_x;
}
```

Each core uses its own private variables and executes this block of code independently of the other cores.

Example (cont.)

- After each core completes execution of the code, a private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.
- Ex., 8 cores, n = 24, each element computes three values, then the calls to Compute_next_value return:

```
1,4,3,  9,2,8,  5,1,1,  5,2,7,  2,5,0,  4,1,8,  6,5,1,  2,3,9
```

Example (cont.)

- Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated “master” core which adds the final result.

Example (cont.)

```
if (I'm the master core) {
    sum = my_x;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_x to the master;
}
```

Example (cont.)

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Global sum

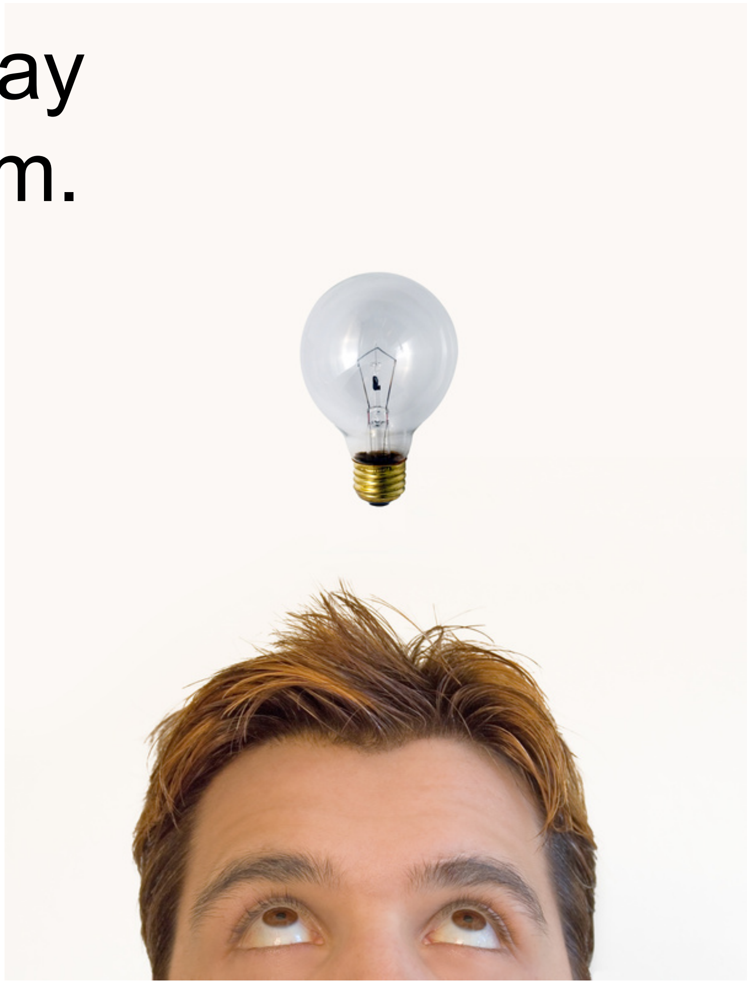
8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

Master core

But wait!

There is a much better way to compute the global sum.



Better parallel algorithm

- Do not make the master core do all the work.
- Share it among the other cores.

1st level

- Pair the cores so that core 0 adds its result with core 1's result.
- Core 2 adds its result with core 3's result, etc.
- Work with odd and even numbered pairs of cores.

Better parallel algorithm (cont.)

- Repeat the process now with only the evenly ranked cores.

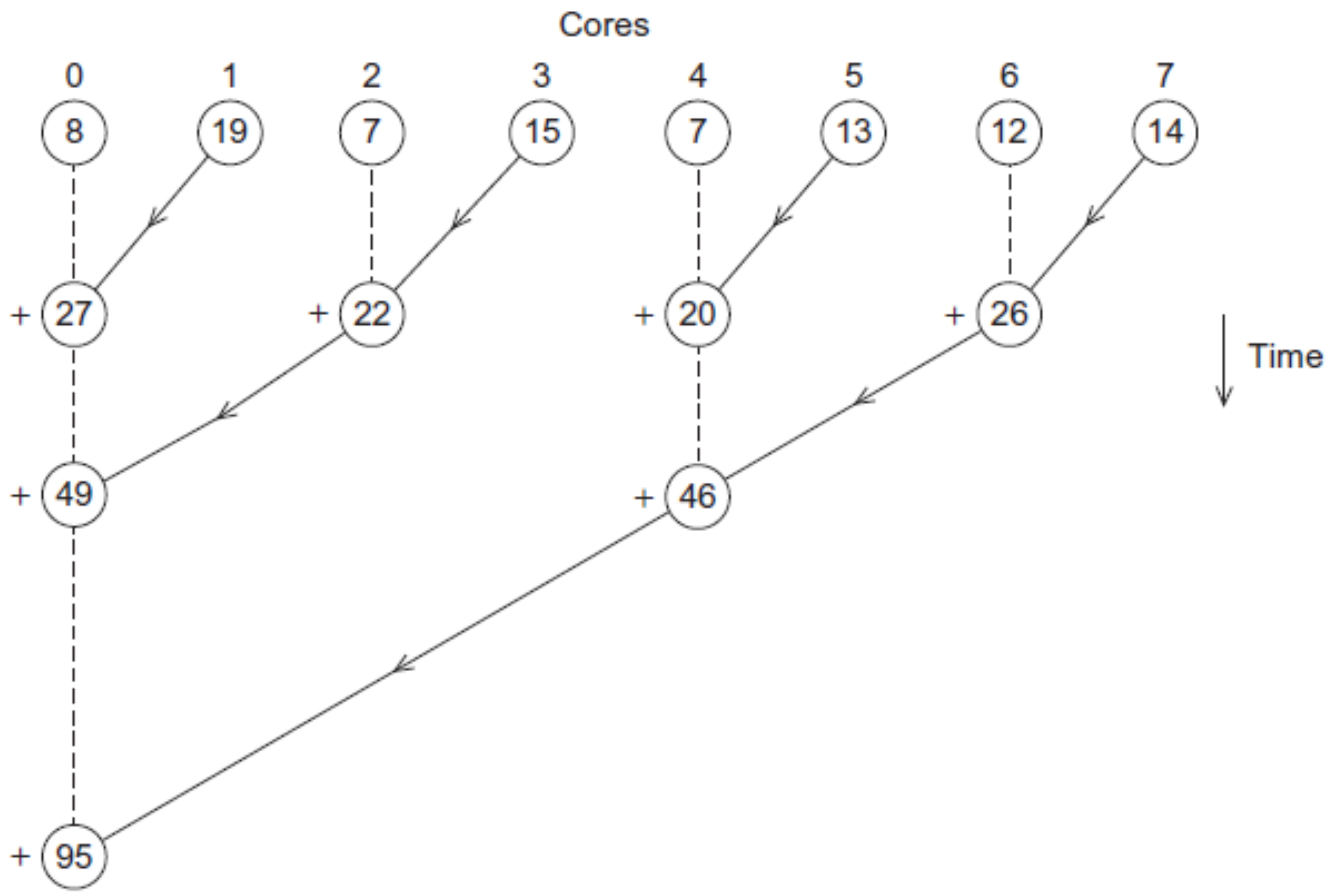
2nd level

- Core 0 adds result from core 2.
- Core 4 adds the result from core 6, etc.

3rd level

- Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.

Multiple cores forming a global sum



1st level

2nd level

3nd level

(…)

Analysis

- In the first example, the master core performs 7 receives and 7 additions.

  Global sum:

  8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

- In the second example, the master core performs 3 receives and 3 additions.

  Global sum:

  8 + 19 + 22 + 46 = 95

- The improvement is more than a factor of 2!

Analysis (cont.)

- The difference is more dramatic with a larger number of cores.
- If we have 1000 cores:
  - The first example would require the master to perform 999 receives and 999 additions.
  - The second example would only require 10 receives and 10 additions.
- That is an improvement of almost a factor of 100!

Roadmap

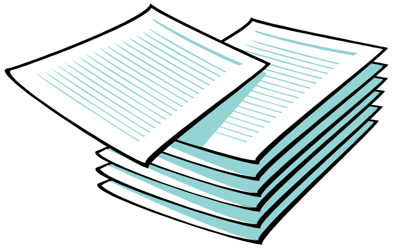
- Why we need ever-increasing performance.
  - More complex problems
- Why we are building parallel systems.
- Why we need to write parallel programs.
- How do we write parallel programs?

How do we write parallel programs?

- Task parallelism
  - Partition various tasks carried out solving the problem among the cores.
- Data parallelism
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on its part of the data.

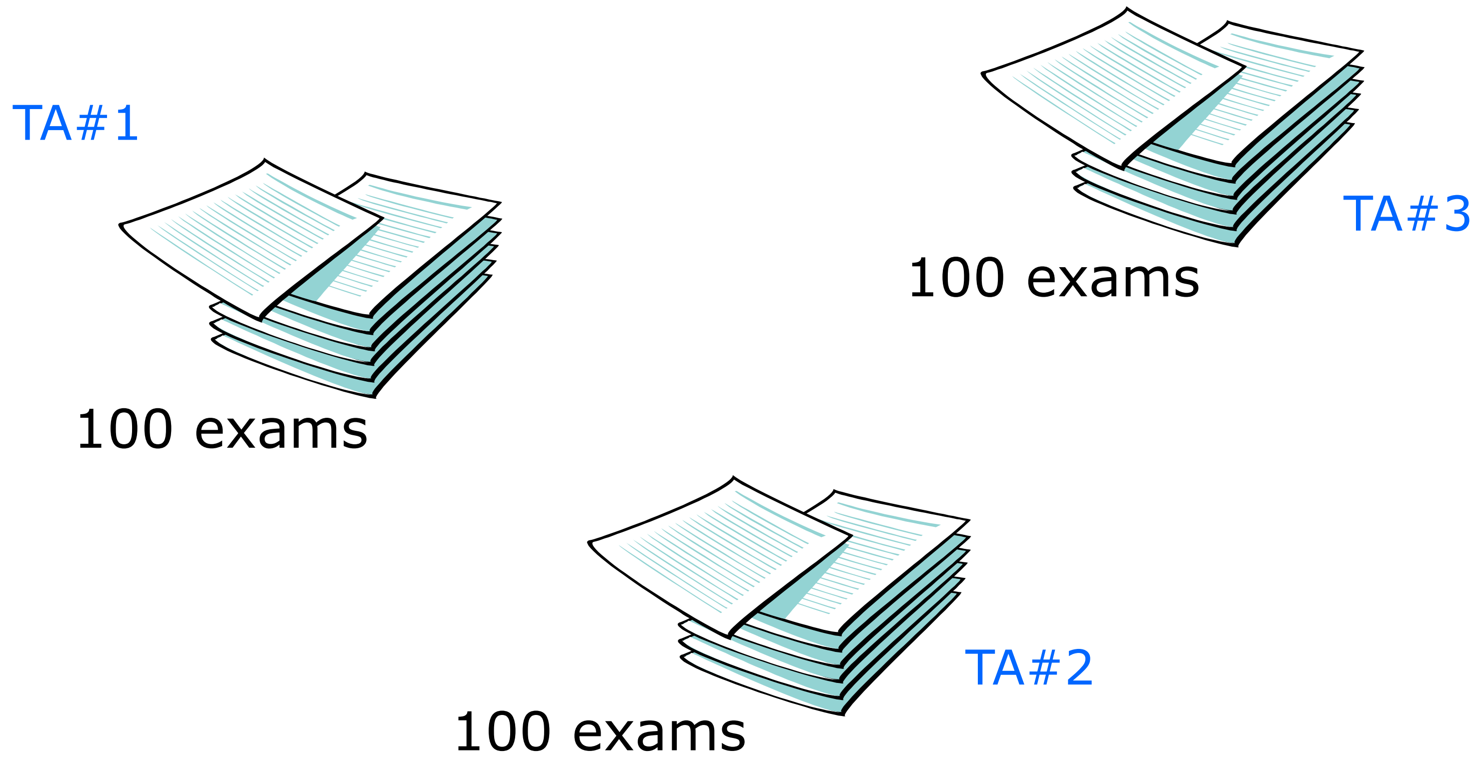
Professor P

15 questions
300 exams



Professor P’s grading assistants

Division of work – data parallelism



Division of work – task parallelism

TA#1

Questions 1 - 5

TA#3

Questions 11 - 15

TA#2

Questions 6 - 10

Division of work – data parallelism

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Division of work – task parallelism

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

Tasks

1) Receiving
2) Addition

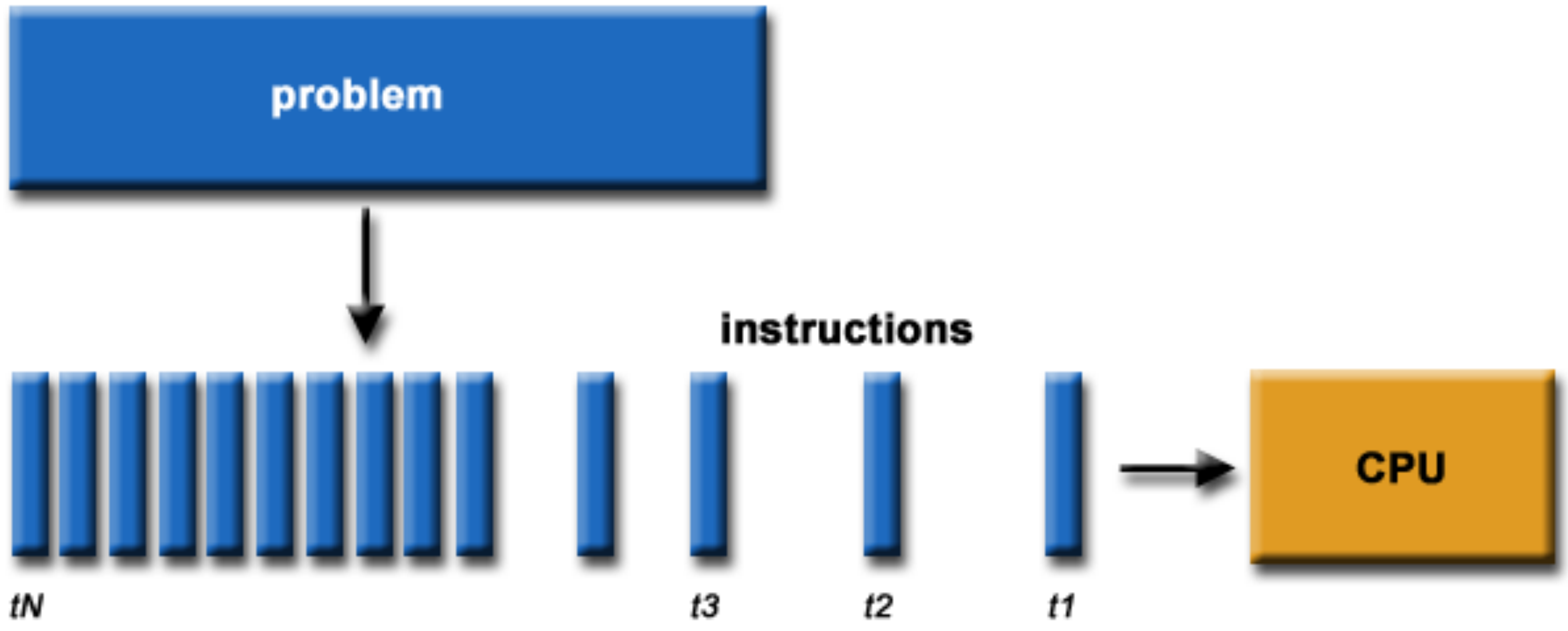
Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core.
- Load balancing – share the work evenly among the cores so that one is not heavily loaded.
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

Illustrating further…

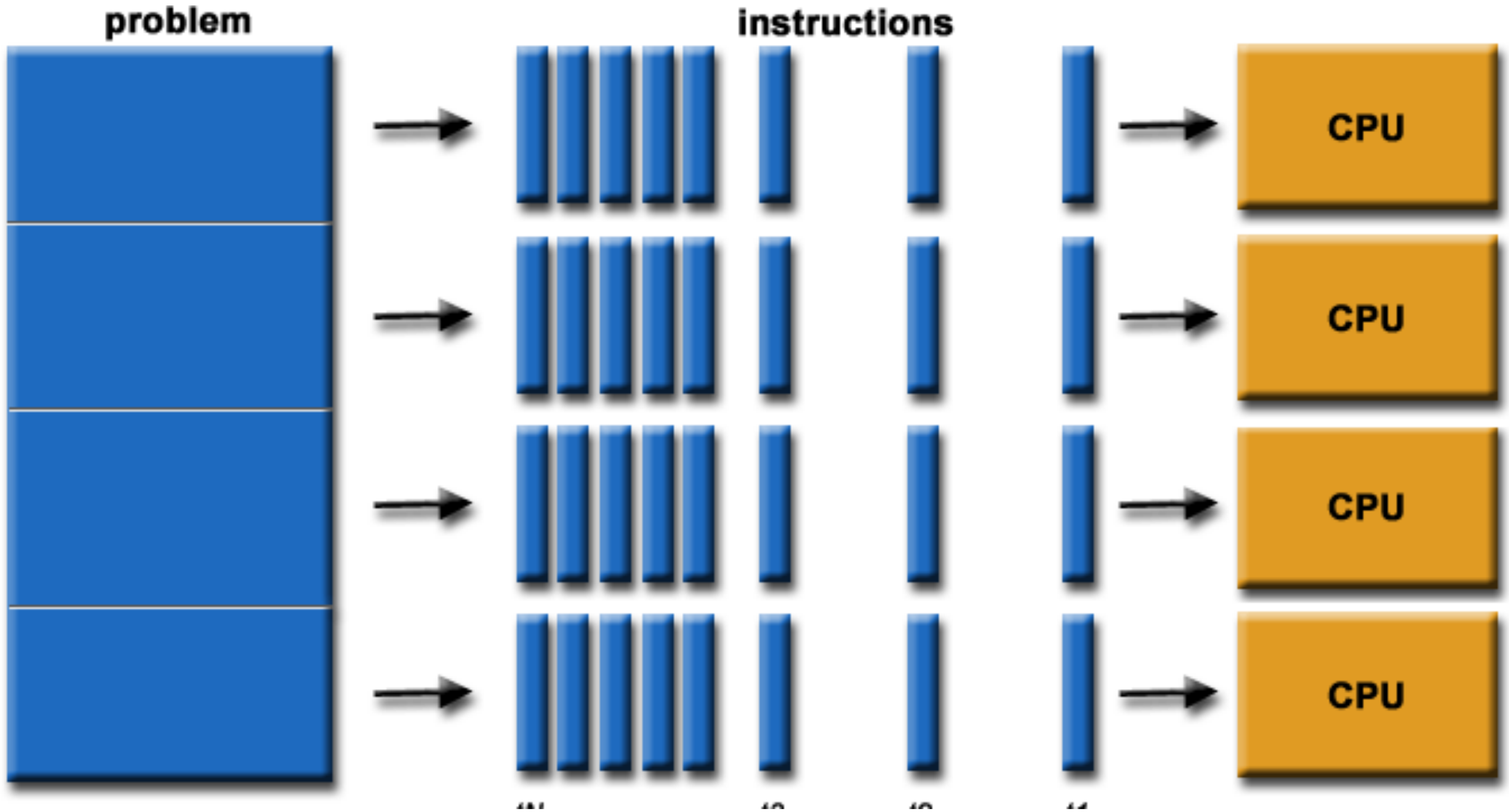
- Sequential computing
- Parallel computing
  - Data parallelism
  - Task parallelism
  - Combining task and data parallelism

Sequential computing



Blaise Barney, LLNL, https://computing.llnl.gov/tutorials/parallel_comp/

Parallel computing



Blaise Barney, LLNL, https://computing.llnl.gov/tutorials/parallel_comp/

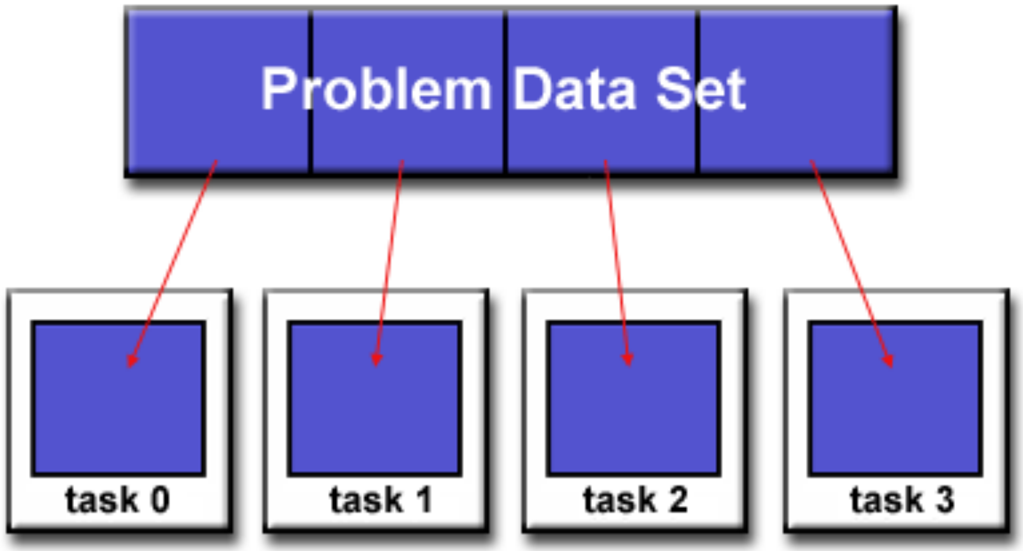
Partitioning/Decomposition

One of the first steps in designing a parallel program:

- break the problem into discrete "chunks" of work that can be distributed to multiple tasks.

Data parallelism – Domain decomposition

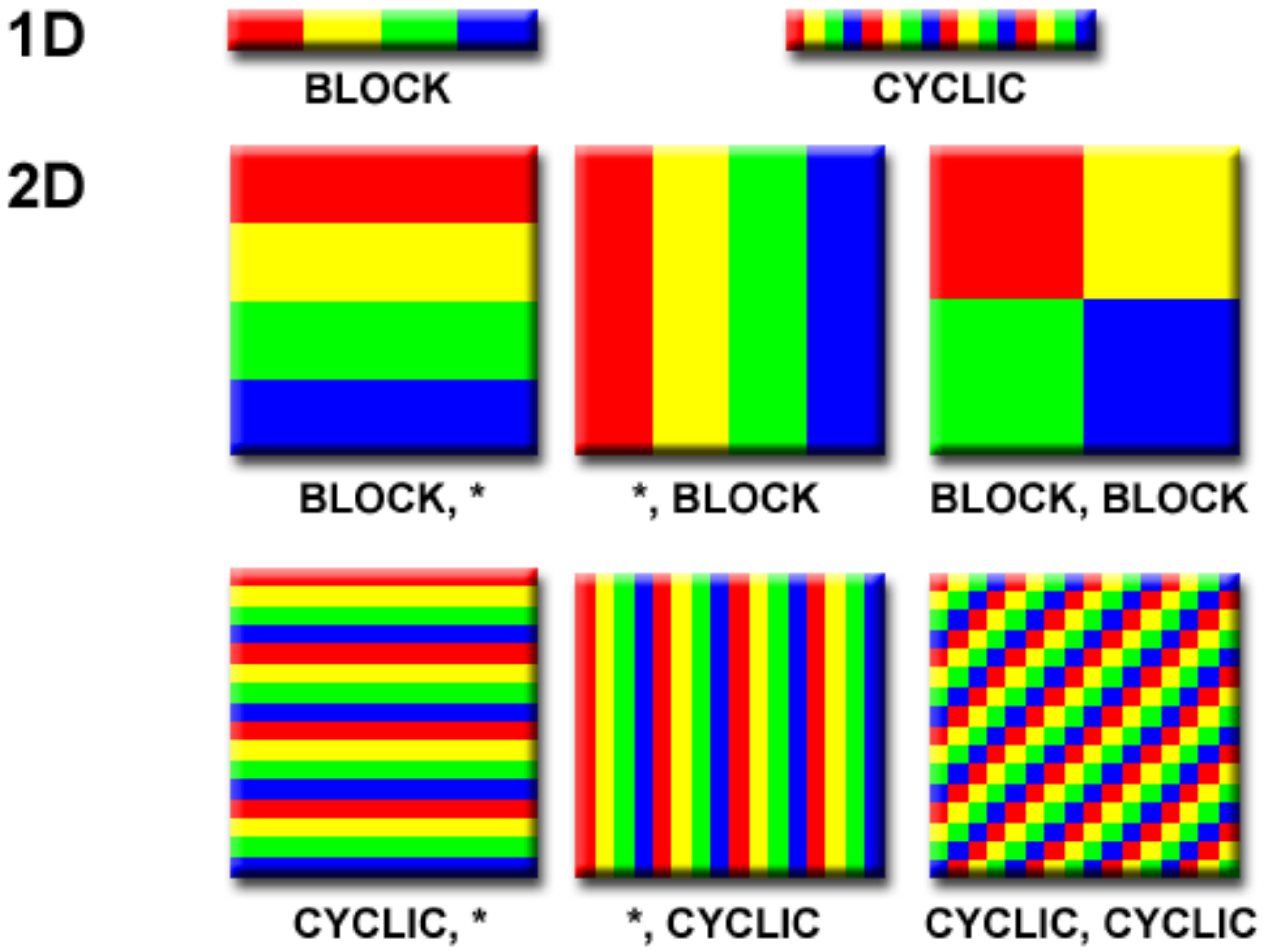
The data associated with a problem is decomposed. Each parallel task then works on a portion of the data.



Blaise Barney, LLNL, https://computing.llnl.gov/tutorials/parallel_comp/

Data parallelism – Domain decomposition

There are different ways to partition data:

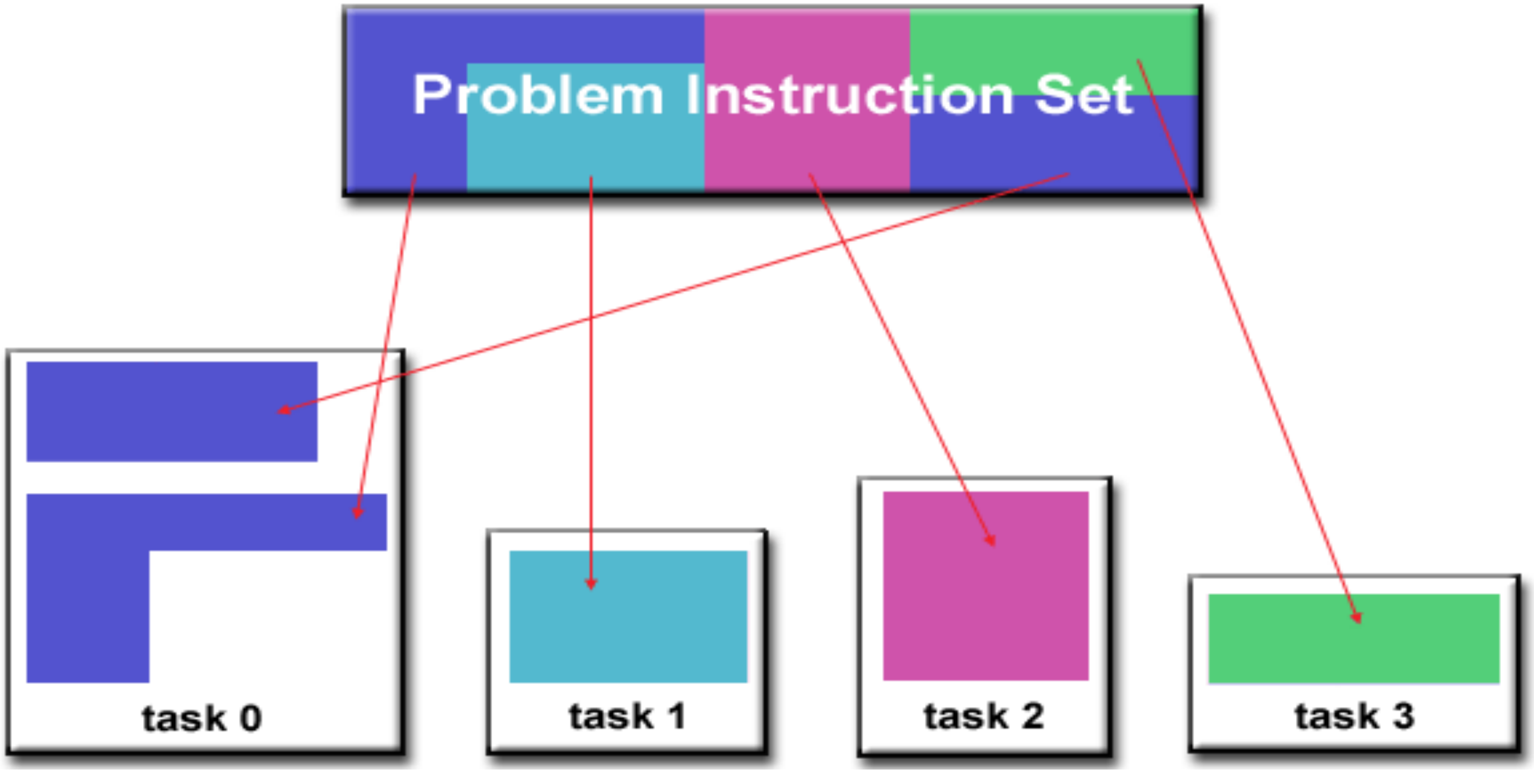


Blaise Barney, LLNL, https://computing.llnl.gov/tutorials/parallel_comp/

Task parallelism – Functional decomposition

The focus is on the computation that is to be performed rather than on the data manipulated by the computation.

The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

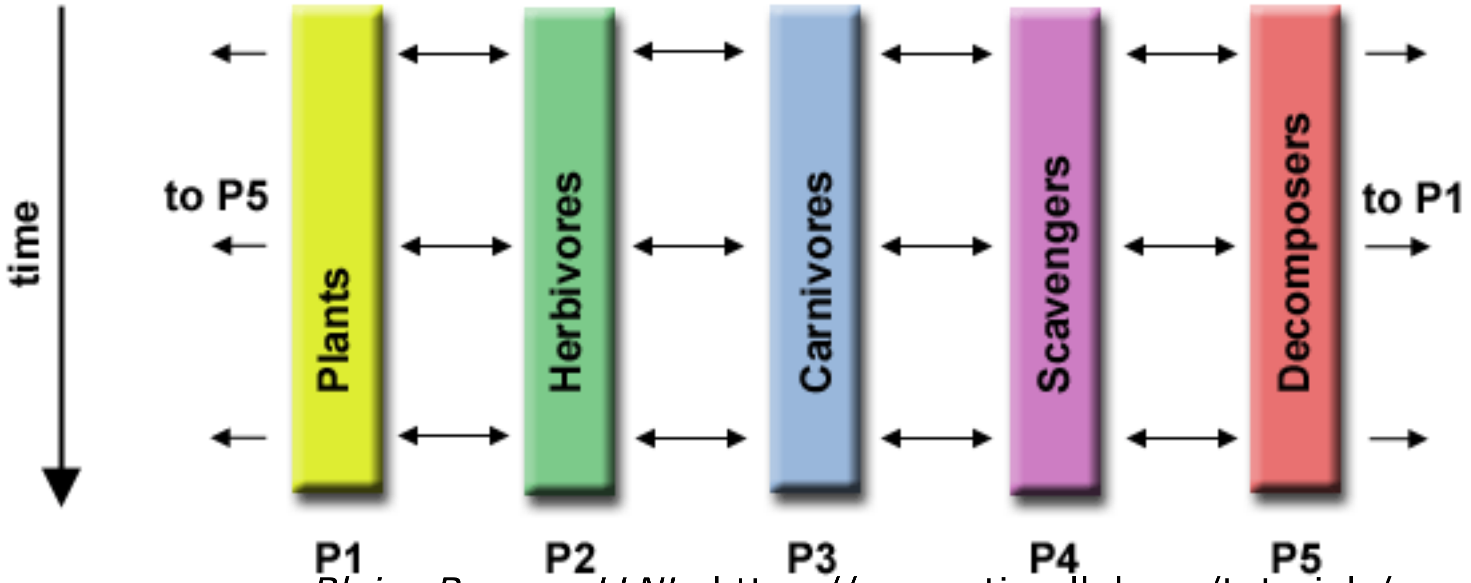


Blaise Barney, LLNL, https://computing.llnl.gov/tutorials/parallel_comp/

Task parallelism – functional decomposition

Functional decomposition lends itself well to problems that can be split into different tasks.

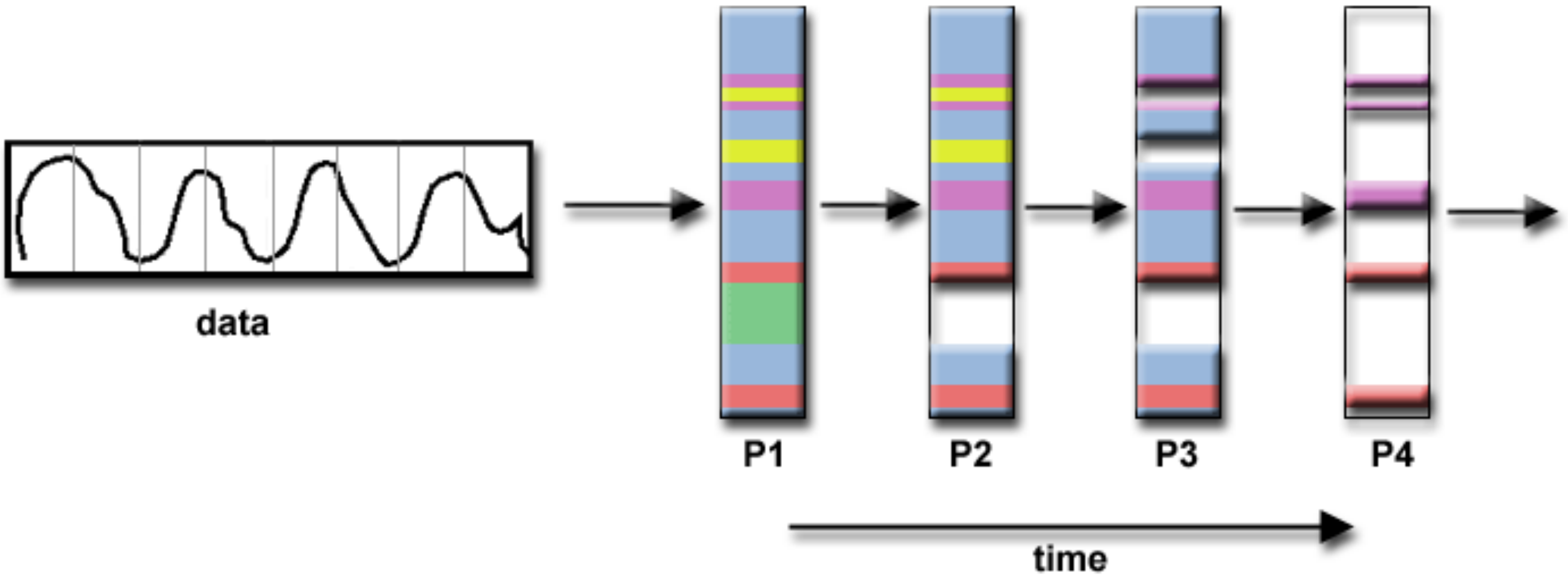
Ecosystem Modeling - Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time step.



Blaise Barney, LLNL, https://computing.llnl.gov/tutorials/parallel_comp/

Task parallelism – functional decomposition

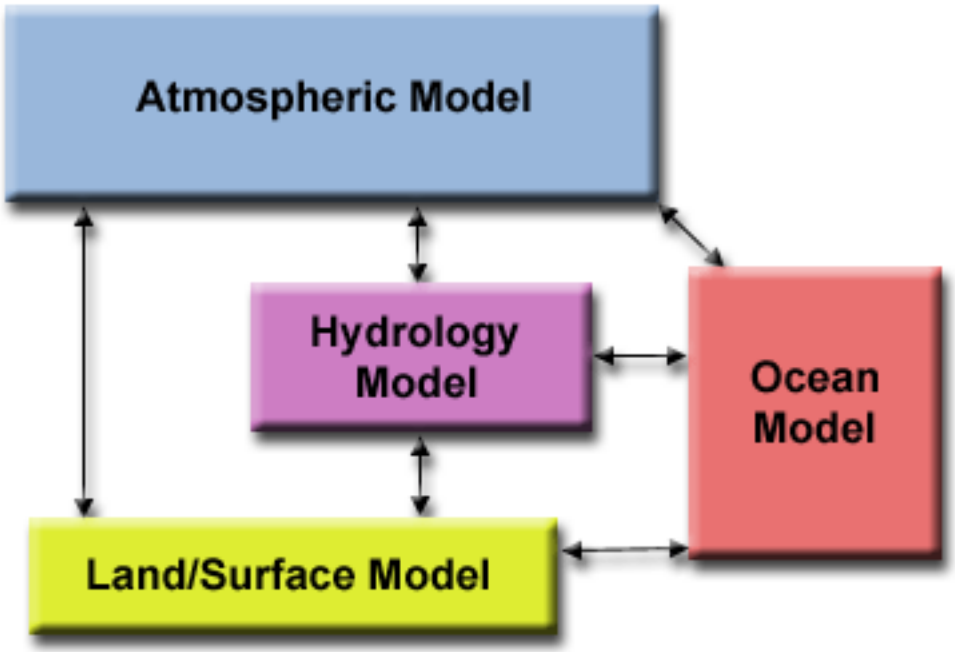
Signal Processing - An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy.



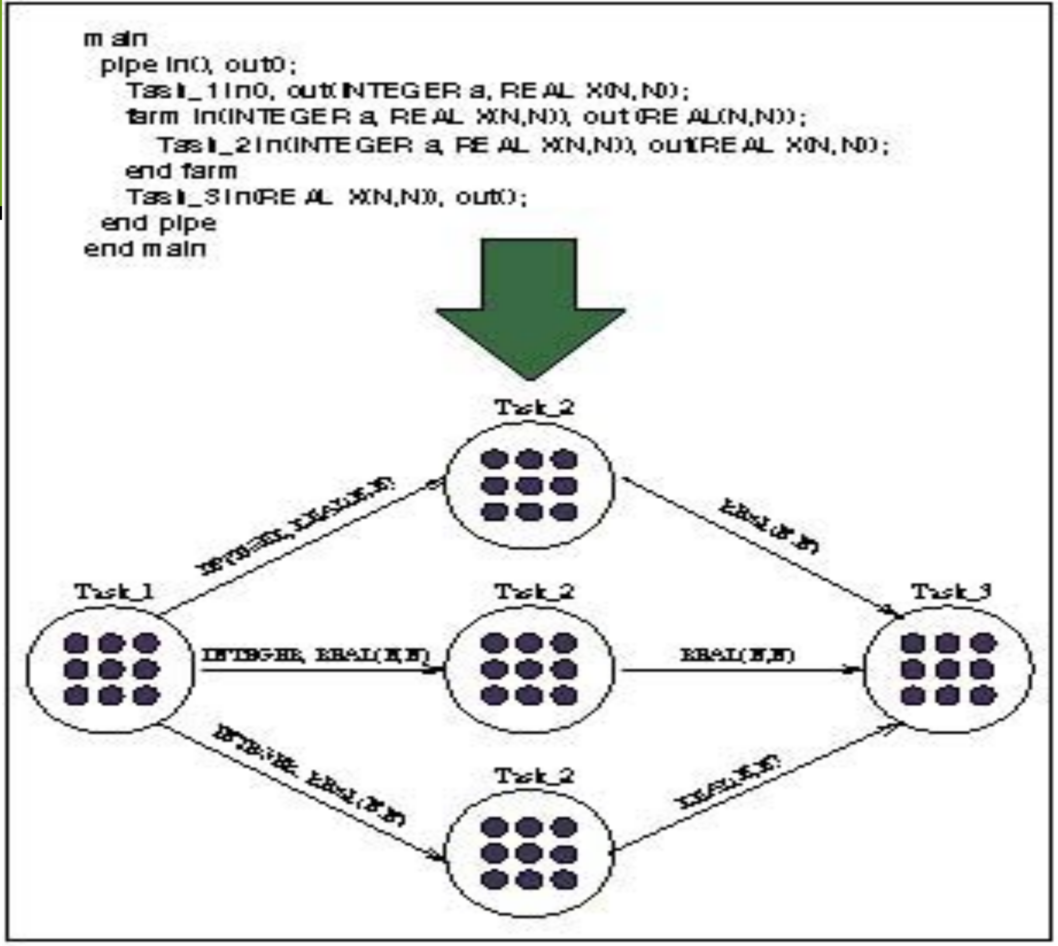
Blaise Barney, LLNL, https://computing.llnl.gov/tutorials/parallel_comp/

Task parallelism – functional decomposition

Climate Modeling- each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model; the ocean model generates sea surface temperature data that are used by the atmosphere model; and so on.



Blaise Barney, LLNL, https://computing.llnl.gov/tutorials/parallel_comp/



```
main
  pipe in(), out();
    Task_1 in(), out(INTEGER a, REAL X(N,N));
    farm in(INTEGER a, REAL X(N,N)), out(REAL(N,N));
      Task_2 in(INTEGER a, REAL X(N,N)), out(REAL X(N,N));
    end farm
    Task_3 in(REAL X(N,N)), out();
  end pipe
end main
```

High-level description and associated task graph of a parallel application obtained by composing three data-parallel tasks according to a pipeline structure where the second stage has been replicated by hierarchically composing the pipeline with a farm structure.

Raffaele Perego and Salvatore Orlando

Combining tasks and data parallelism is common.

Reusing knowledge on parallel programming

- There is an extensive body of knowledge on how to decompose a problem, e.g.
  - Data parallelism
  - Task parallelism
- Decomposition schemes are recurrent across several distinct problems
- Reuse that knowledge of experts – e.g. parallel patterns in textual descriptions
  - Build a common language and transfer knowledge in small "chunks"
- Make that knowledge easily reusable
  - E.g. Parametrisation of parallel abstractions like algorithmic skeletons available in frameworks

Roadmap

- Why we need ever-increasing performance.
  - More complex problems
- Why we' re building parallel systems.
- Why we need to write parallel programs.
- How do we write parallel programs?
- What we will be doing.

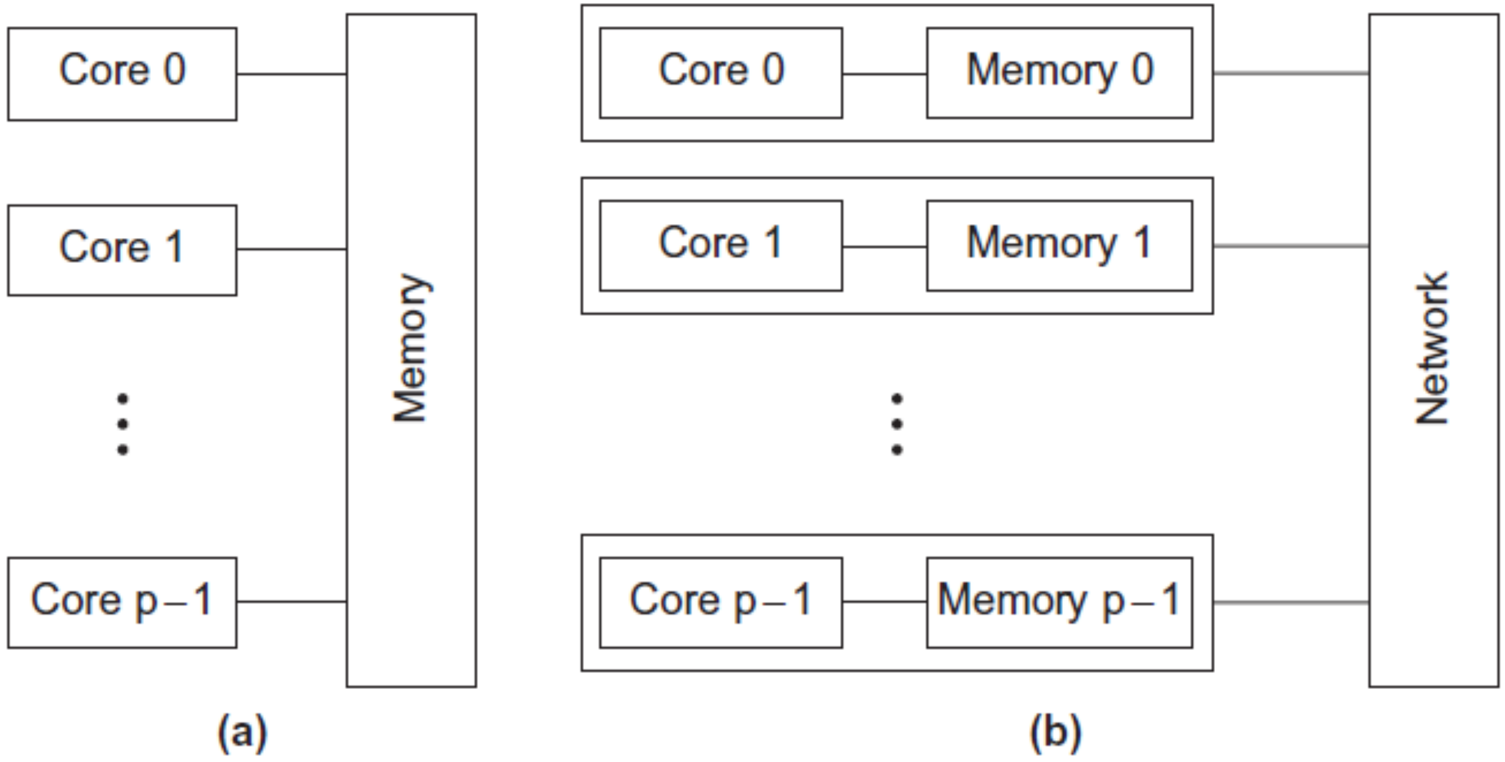
What we' ll be doing

- Learning to write programs that are explicitly parallel.
  - C (C++) languages.
- Using different extensions to C (C++).
  - Posix Threads (Pthreads)
  - OpenMP
  - CUDA (Compute Unified Device Architecture) – NVIDIA GPUs
  - Message-Passing Interface (MPI)
- Identification of common parallel patterns

Type of parallel systems

- Shared-memory
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- Distributed-memory
  - Each core has its own, private memory.
  - The cores must communicate explicitly by sending messages across a network.

Type of parallel systems



Shared-memory Distributed-memory

Terminology

- Concurrent computing – a program is one in which multiple tasks can be in progress at any instant.
- Parallel computing – a program is one in which multiple tasks cooperate closely to solve a problem
- Distributed computing – a program may need to cooperate with other programs to solve a problem.

Concluding Remarks (1)

- The laws of physics have brought us to the doorstep of multicore technology.
- Serial programs typically do not benefit from multiple cores.
- Automatic parallel program generation from serial program code is not the most efficient approach to get high performance from multicore computers.

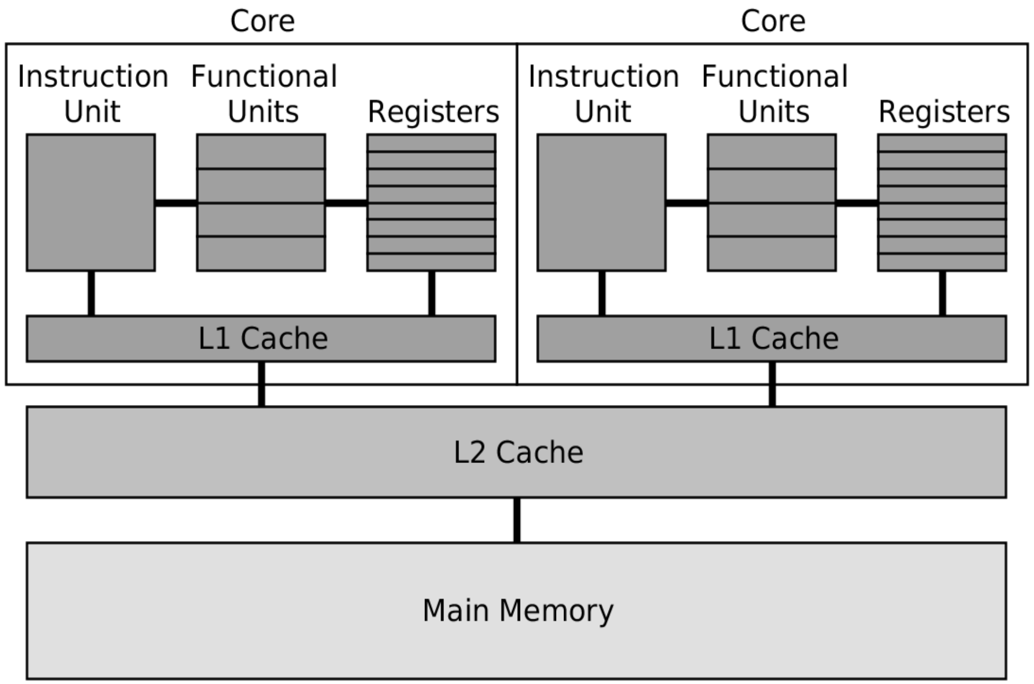
Concluding Remarks (2)

- Learning to write parallel programs involves learning how to coordinate the cores.
- Parallel programs are usually very complex and therefore, require sound program techniques and development
  - simplify programmers' effort, whenever possible…

Topics

1. Parallel computing: hardware, software, applications and performance theory. (2 weeks)

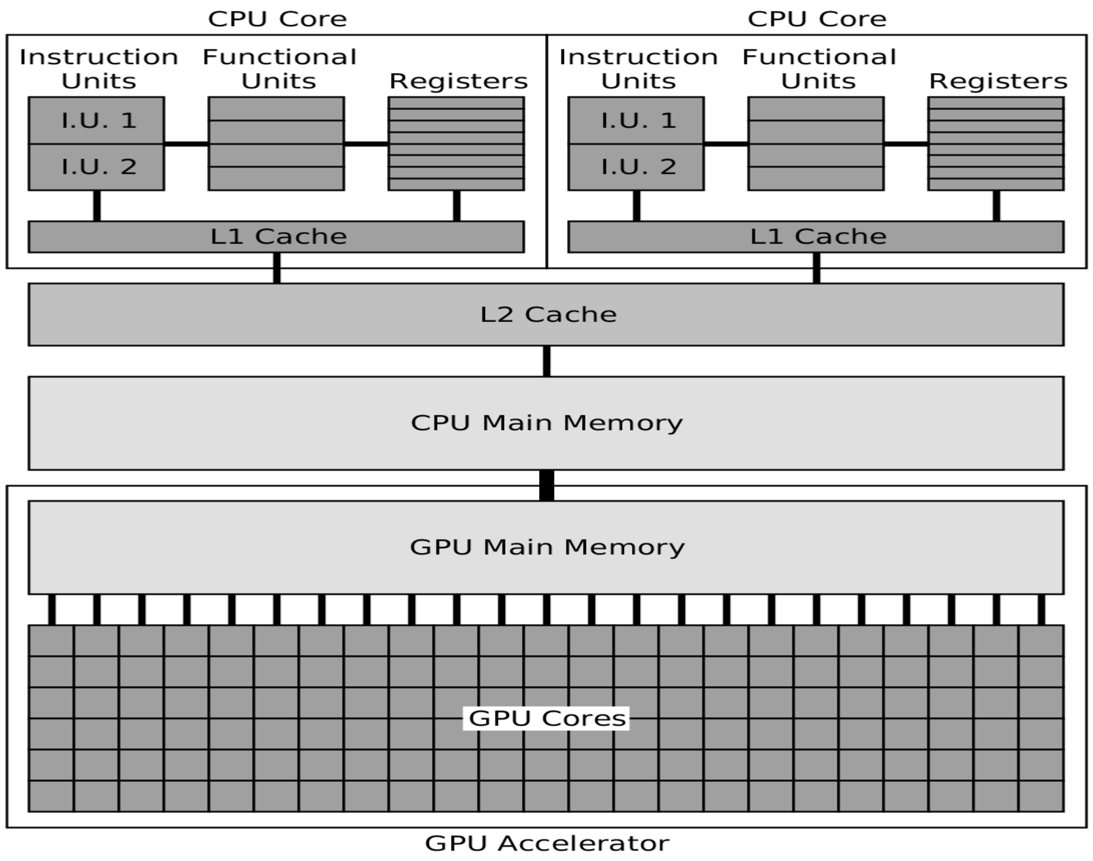
2. Programming shared-memory multiprocessors. OpenMP. Application examples. (3 weeks)



Courtesy:
Alan Kaminsky

Topics

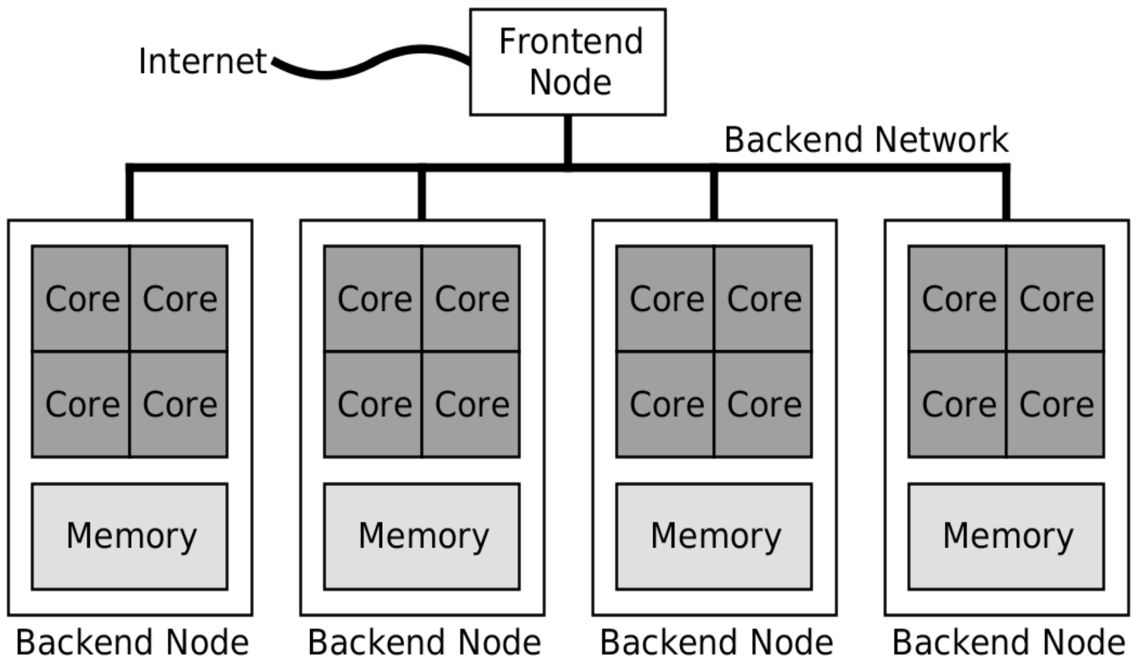
3. GPU computing. CUDA (3 weeks)



Courtesy:
Alan Kaminsky

Topics

4. Programming distributed-memory multiprocessors. MPI. Application examples. (3 weeks)

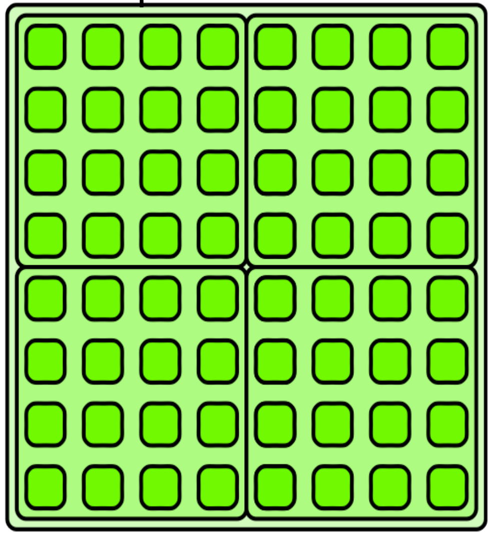


Courtesy:
Alan Kaminsky

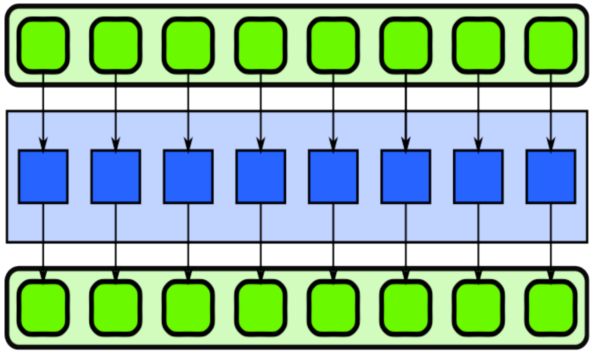
Topics

5. Structured Parallel Programming. (1 weeks / included in the other modules)

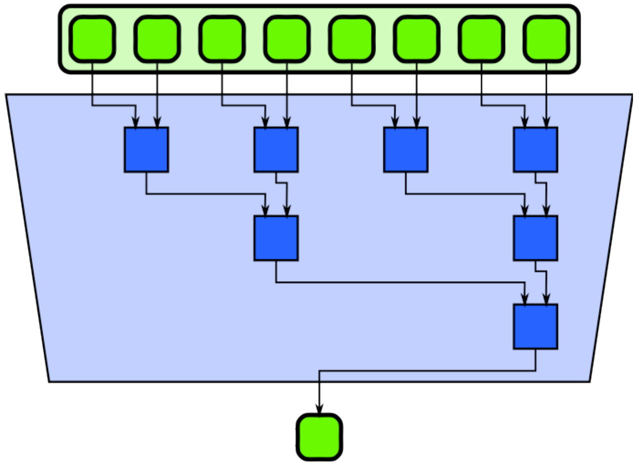
Geometric Decomposition



Map



Reduce



Courtesy: McCool, Robison, and Reinders

References

- Chapter 1 of Lin & Snyder, “Principles of Parallel Programming”, Pearson Int Ed., 2009
- Chapter 1 of P. Pacheco, " An Introduction to Parallel Programming", Morgan Kauffman, 2011