ITMO UNIVERSITY

Parallel Algorithms of Data Analysis and Synthesis (PA)

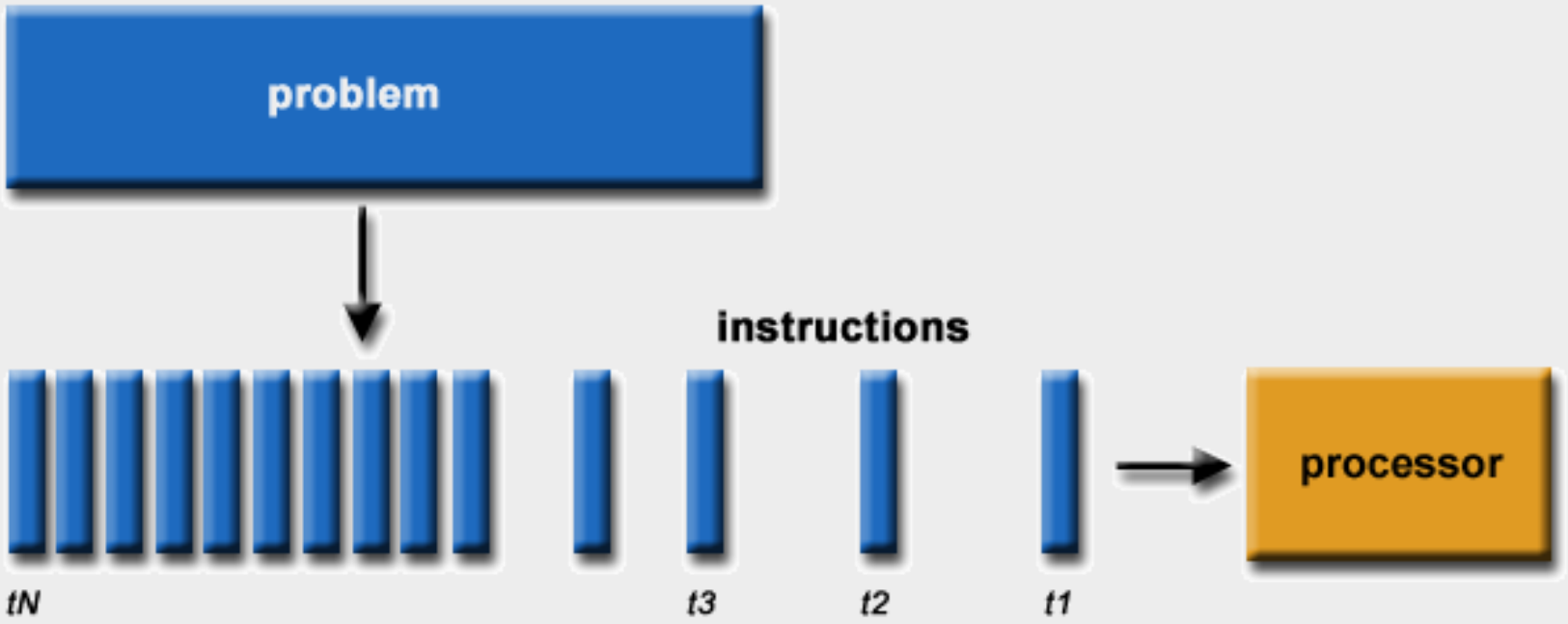
Introduction

PhD Katerina Bolgova

Ekaterina_bolgova@itmo.ru

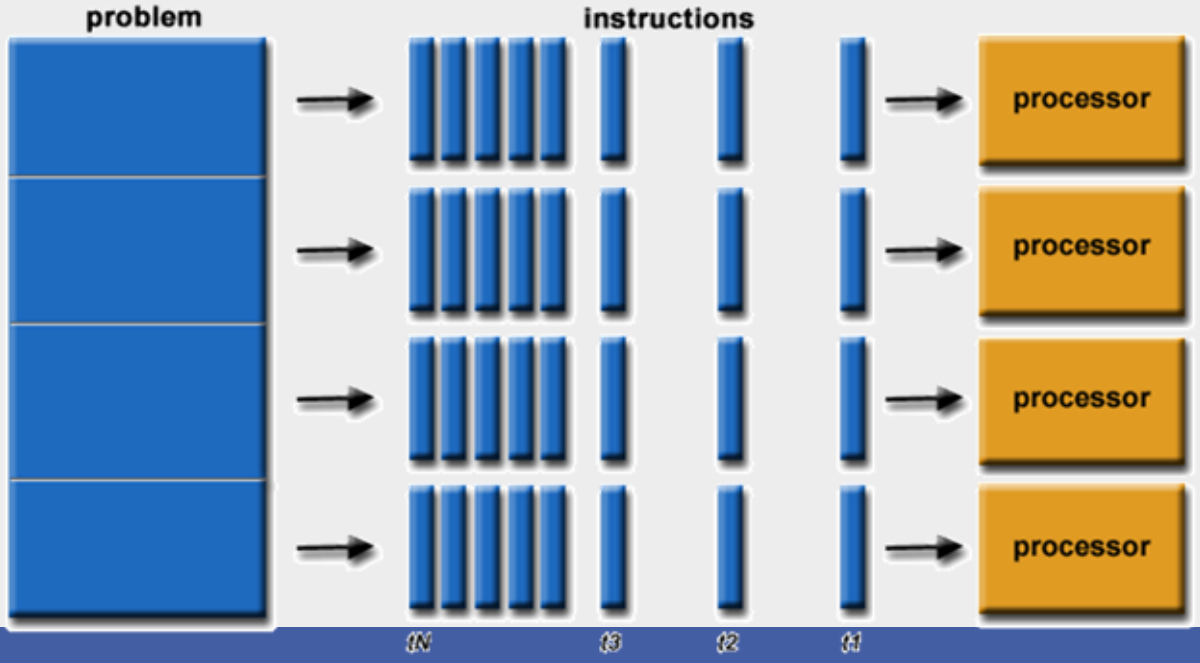
Serial Computing

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Instruction is executed on a single processor
- Only one instruction may be executed at any moment in time



What is Parallel Computing?

- A problem is broken into discrete parts to be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part are executed simultaneously on different processors
- An overall control/coordination mechanism is employed



The First Definition

A *parallel computer*

is a set of processing elements that are able to work cooperatively to solve computational problems quickly.

computers, processors, nodes, cores…

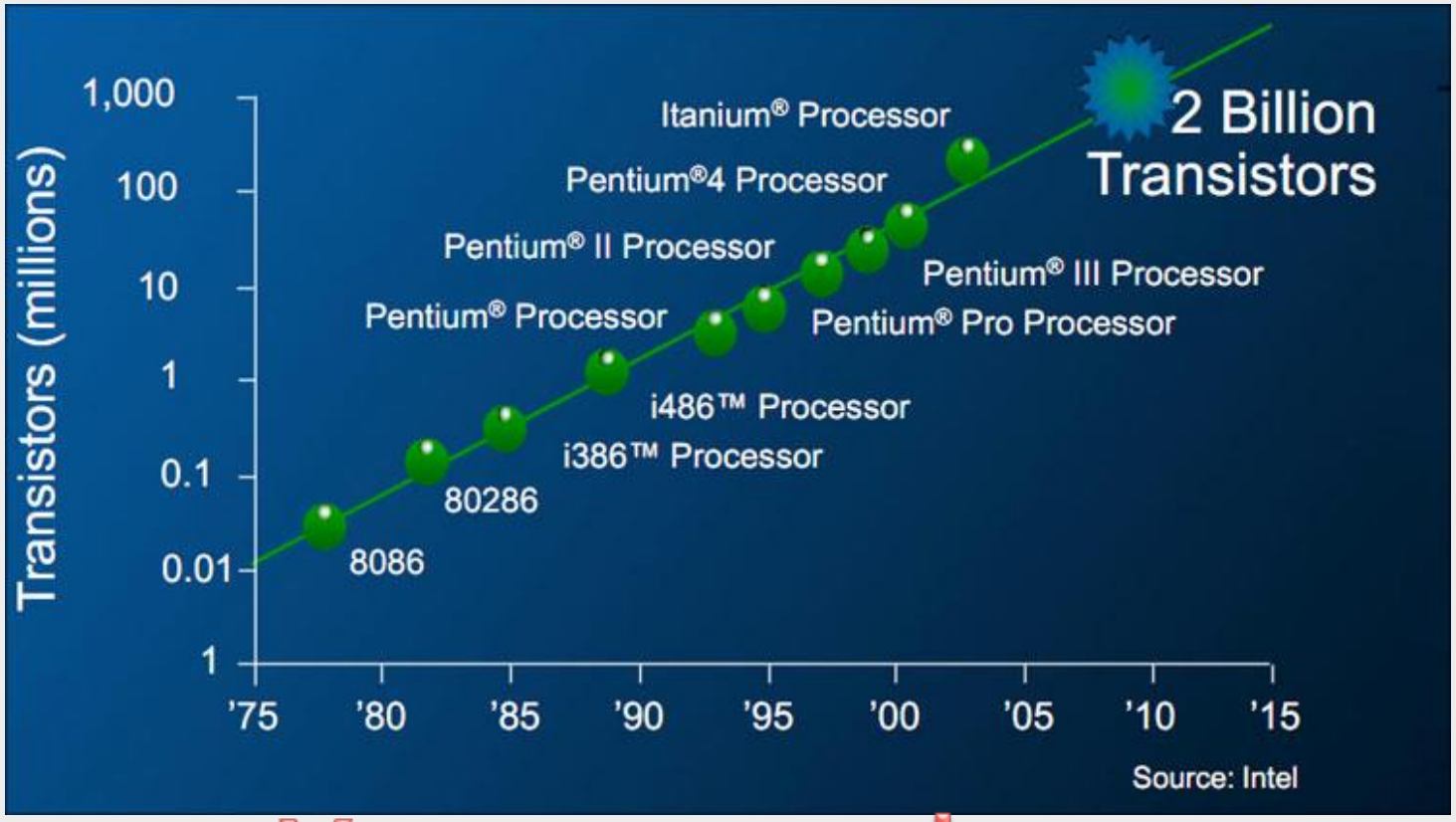
We should care about performance and efficiency

Parallel computing is using multiple processors in parallel to solve problems faster than with a single processor

Processor Development

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months (**"Moore's Law")**

Microprocessors have become smaller, denser, and more powerful!

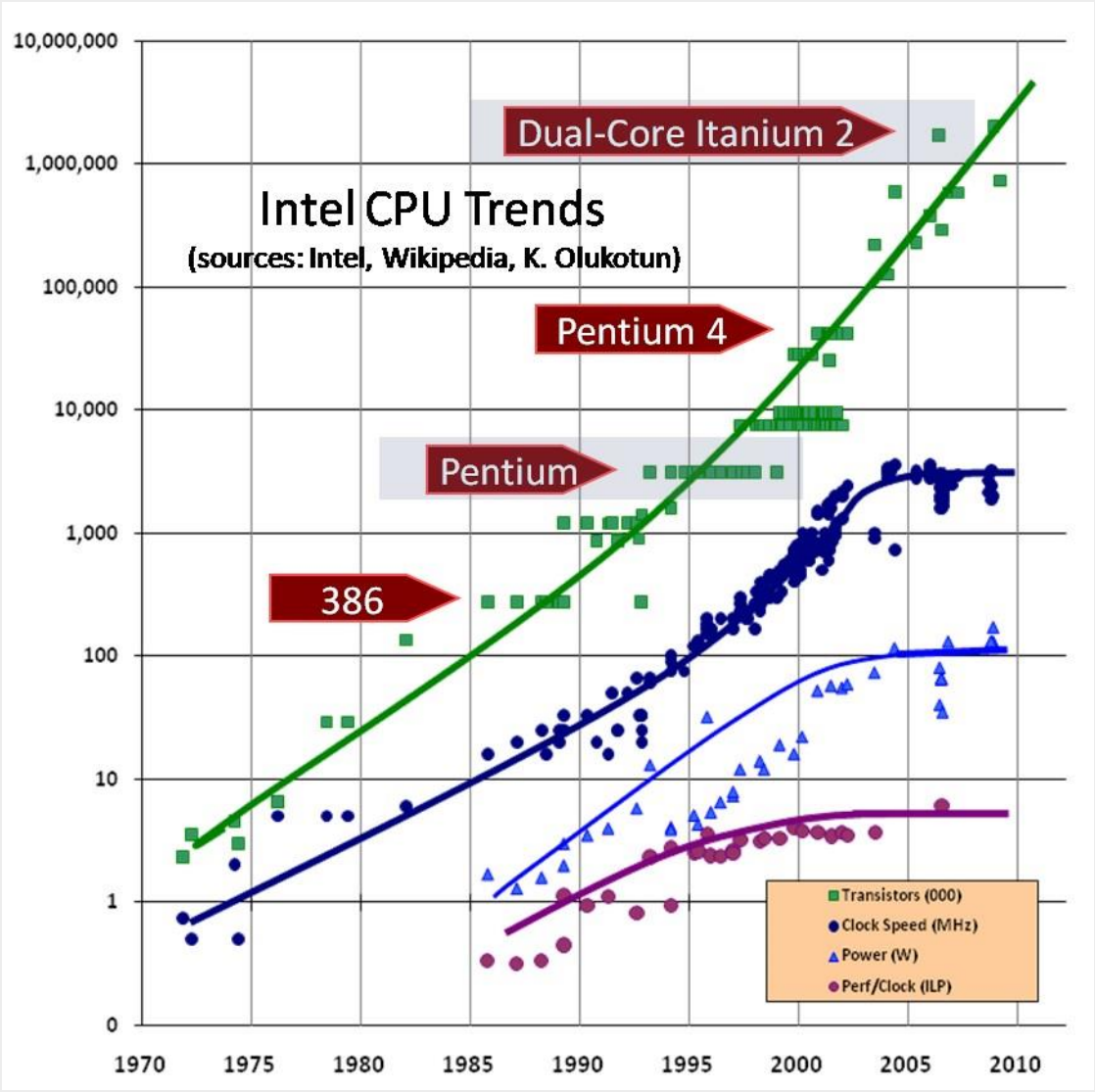


You can run, but...

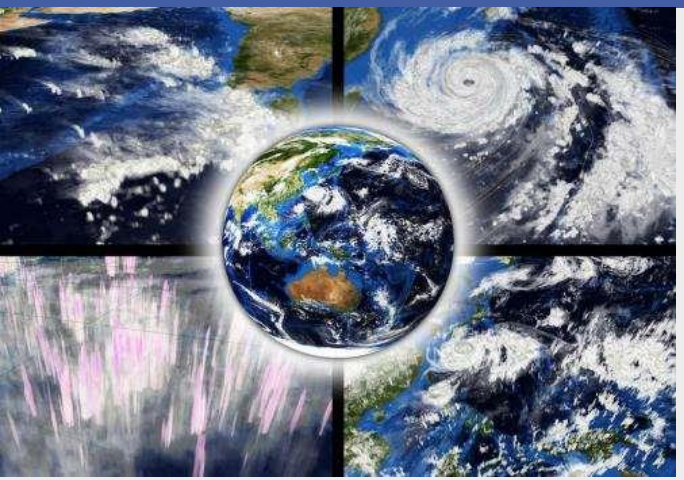
Processor Development (cont.)

… but you can't hide

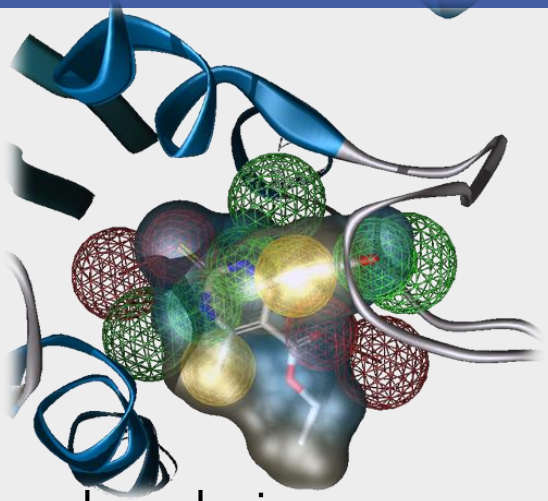
- Clock rates stopped increasing in ~2005 because of limits on power consumption
- More power = More heat
- Technological limits



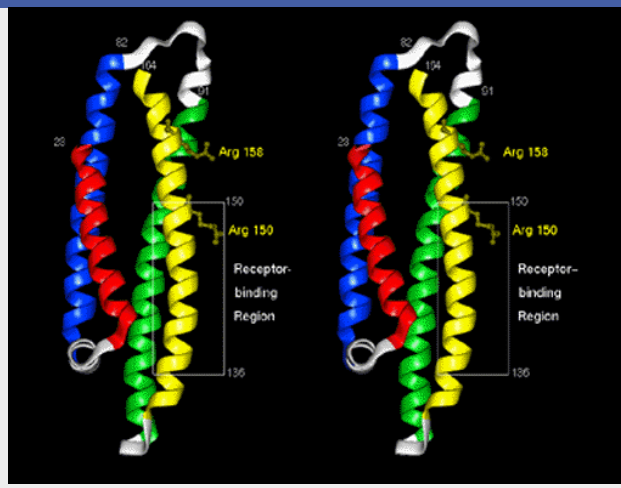
Who Needs Computational Power?



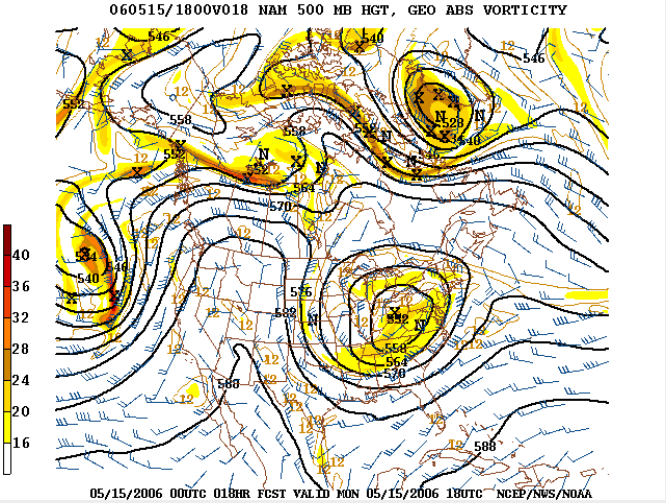
- climate change



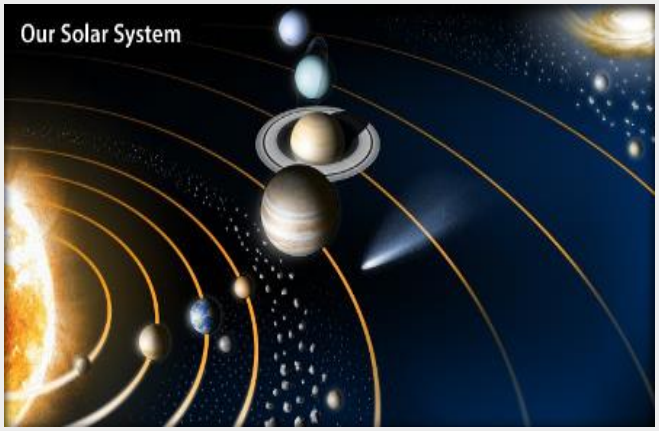
- drug design



- modeling of the human genome



- weather forecasting



- planetary movements

- rush hour traffic

Why Parallel Computing?

- Moore's law isn't true any longer: It's impossible to make reliable low-cost processors that run significantly faster.

- Lots of computer systems that we use every day are parallel systems.
- If we want to take advantage of the processing power of these systems, we need parallel computing
- Parallel computing makes it possible to
  - solve problems that are too large to fit in the memory of one processor
  - solve problems that take too long to solve on a single processor

Parallel Computer Memory Architectures

- SHARED MEMORY
- DISTRIBUTED MEMORY
- HYBRID DISTRIBUTED-SHARED MEMORY

Shared Memory

- Multiple processors can operate independently but share the same memory resources
- Changes in a memory location effected by one processor are visible to all other processors
- Shared memory machines have been classified as UMA and NUMA

Advantages:

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

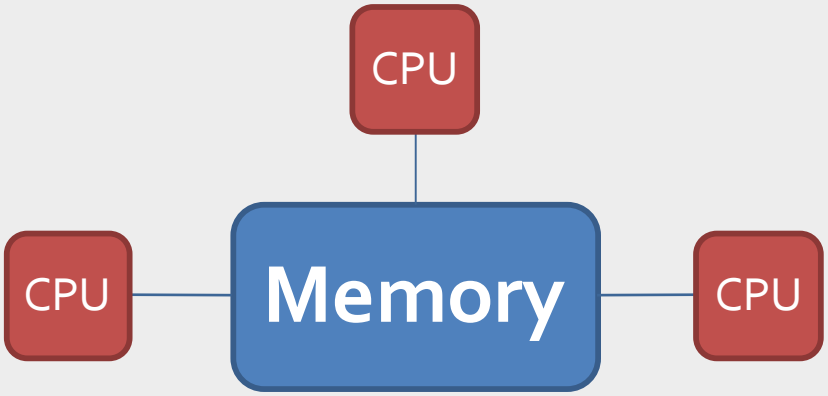
Disadvantages:

- Lack of scalability between memory and CPUs
- Programmer is responsible for **Synchronization Constructs** that ensure *correct* access of global memory

Shared Memory

Uniform Memory Access (UMA):

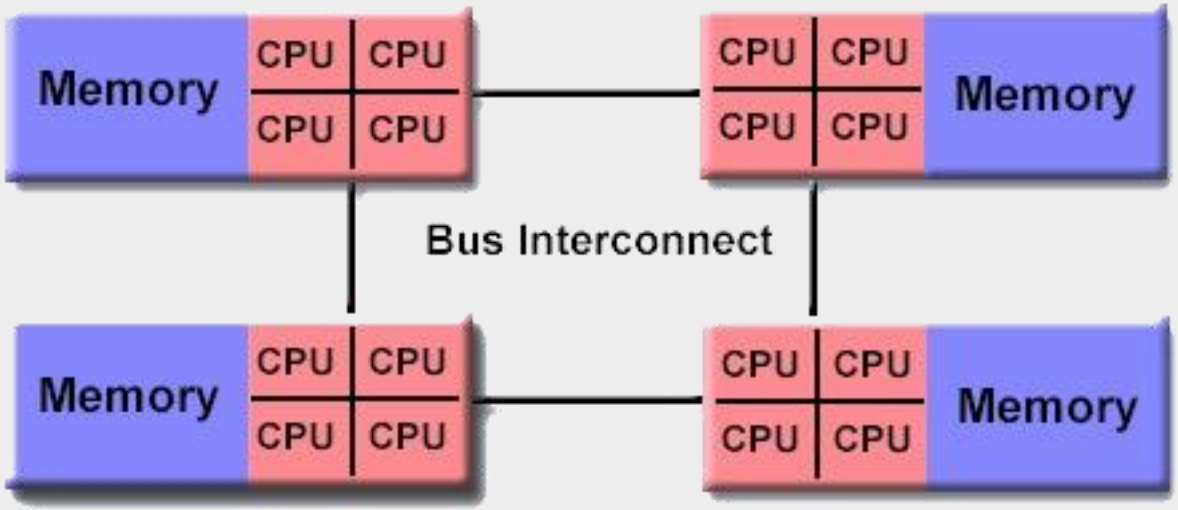
- Most commonly represented today by ***Symmetric Multiprocessor (SMP)*** machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update.



Shared Memory

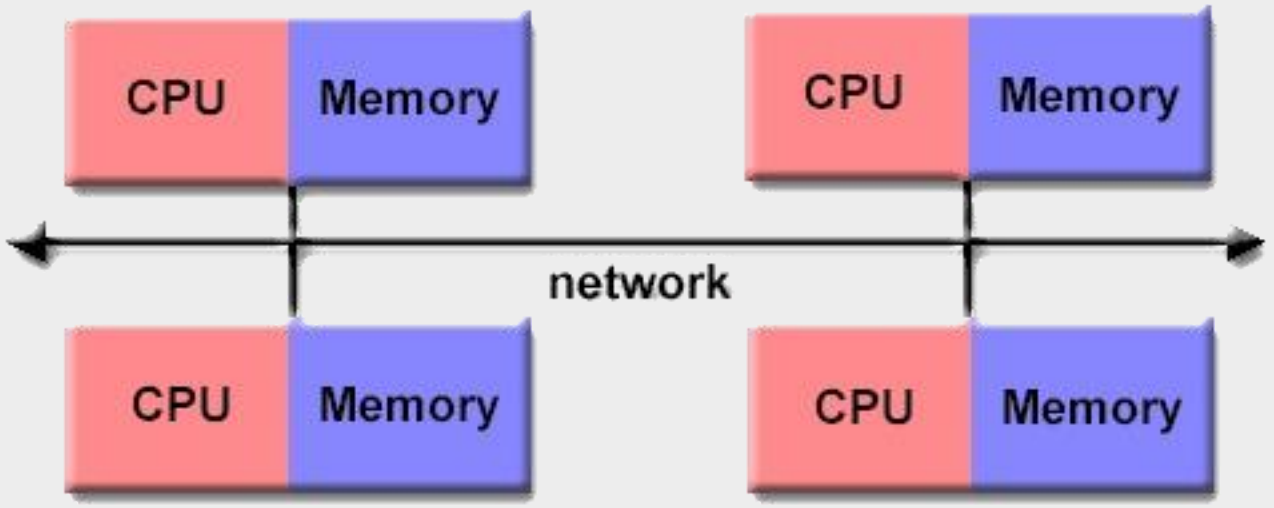
Non-Uniform Memory Access (NUMA):

- Often made by linking physically two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA



Distributed Memory

- The processors are complete computer systems
  - each processor has its own local memory and address space
  - connected to each other by a communication network
- It is usually the task of the programmer to explicitly define how and when data is communicated
- Synchronization between tasks is likewise the programmer's responsibility



Distributed Memory

Advantages:

- Memory is scalable with the number of processors.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency.
- Cost *effectiveness*: can use commodity, off-the-shelf processors and networking.

Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.

Speedup

Parallel computing is using multiple processors in parallel to solve problems more quickly ? than with a single processor

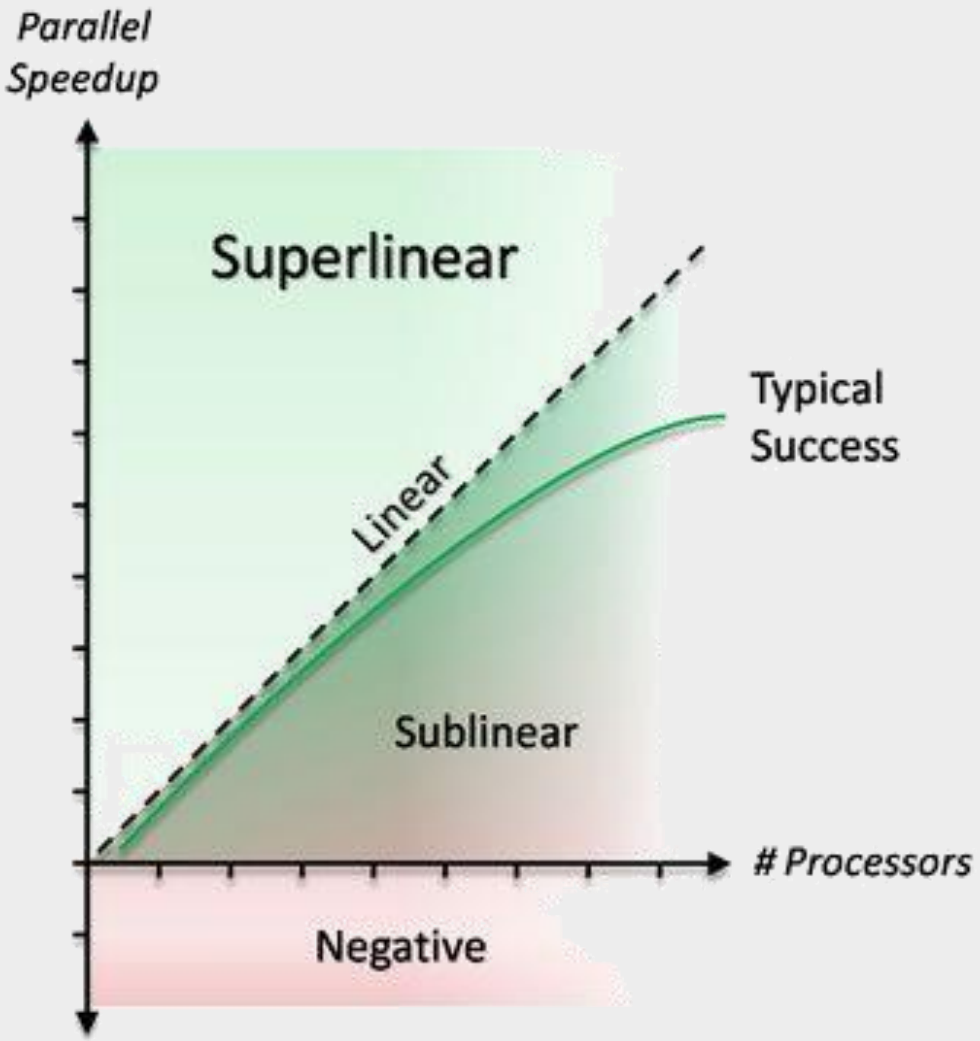
- define **Speedup S** as:
  - the ratio of 2 program execution times
  - constant problem size

$$S_p = \frac{T_1}{T_p}$$

- T_1 is the execution time for the problem on a single processor
  - ***Absolute Speedup*** if this is measured with the *best* serial implementation
  - ***Relative Speedup*** if we use the parallel implementation with one CPU
    - remember that algorithms probably change when moving to parallel
- T_P is the execution time for the problem using P processors

Speedup

- **Linear (ideal) speedup**
  - the time to execute the problem decreases by the number of processors
  - if a job requires 1 week with 1 processor, it will take less that 10 minutes with 1024 processors
- **Sublinear speedup**
  - the most often speedup
- **Superlinear speedup** ?



It's Not Easy to Make it Fast

$$T = T_p + T_c + T_i$$

Three contributions to the total time spend in the program:

- ***Parallelism phase***: time spend managing parallel threads, overhead of synchronization ($T_{p)}$
- ***Computation phase***: time spend calculating on independent, local data ($T_{c)}$
- ***Interaction phase***: time spend in communication, synchronization or other data movement (T_i)

Overhead of Parallelism

- Load imbalance
  - some processes may be idle while the others are working

  - start-up and shutdown phases
  - inherently sequential parts of the computation, which can not be done in parallel

  - computations may not be evenly distributed among the processors (load balancing)

Overhead of Parallelism

- Algorithmic overhead
  - extra computations in the parallel solution which are not present in the sequential solution
  - parallel algorithm may have to do more work to keep distributed data structures up to date
  - calculations may be duplicated in order to avoid communication

- Communication time
  - all communication in a parallel program is the overhead caused by the parallel decomposition, compared to a sequential solution

Measuring speedup is not always easy

- Memory limitations

- Cache effects

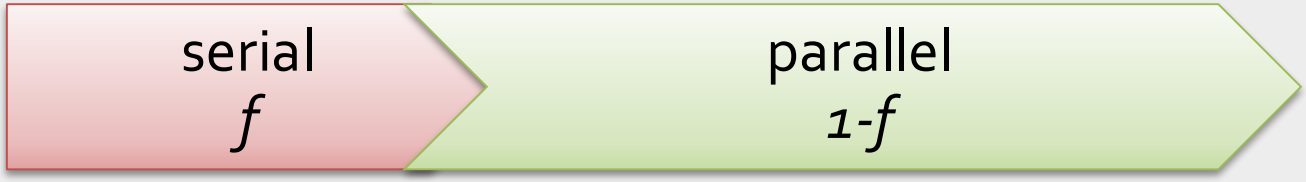
- For measuring absolute speedup, we have to implement both a sequential and a parallel algorithm

- The best sequential algorithm may not be known or may change over time

- Using faster processors in a parallel solution may result in a smaller speedup

Amdahl's Law

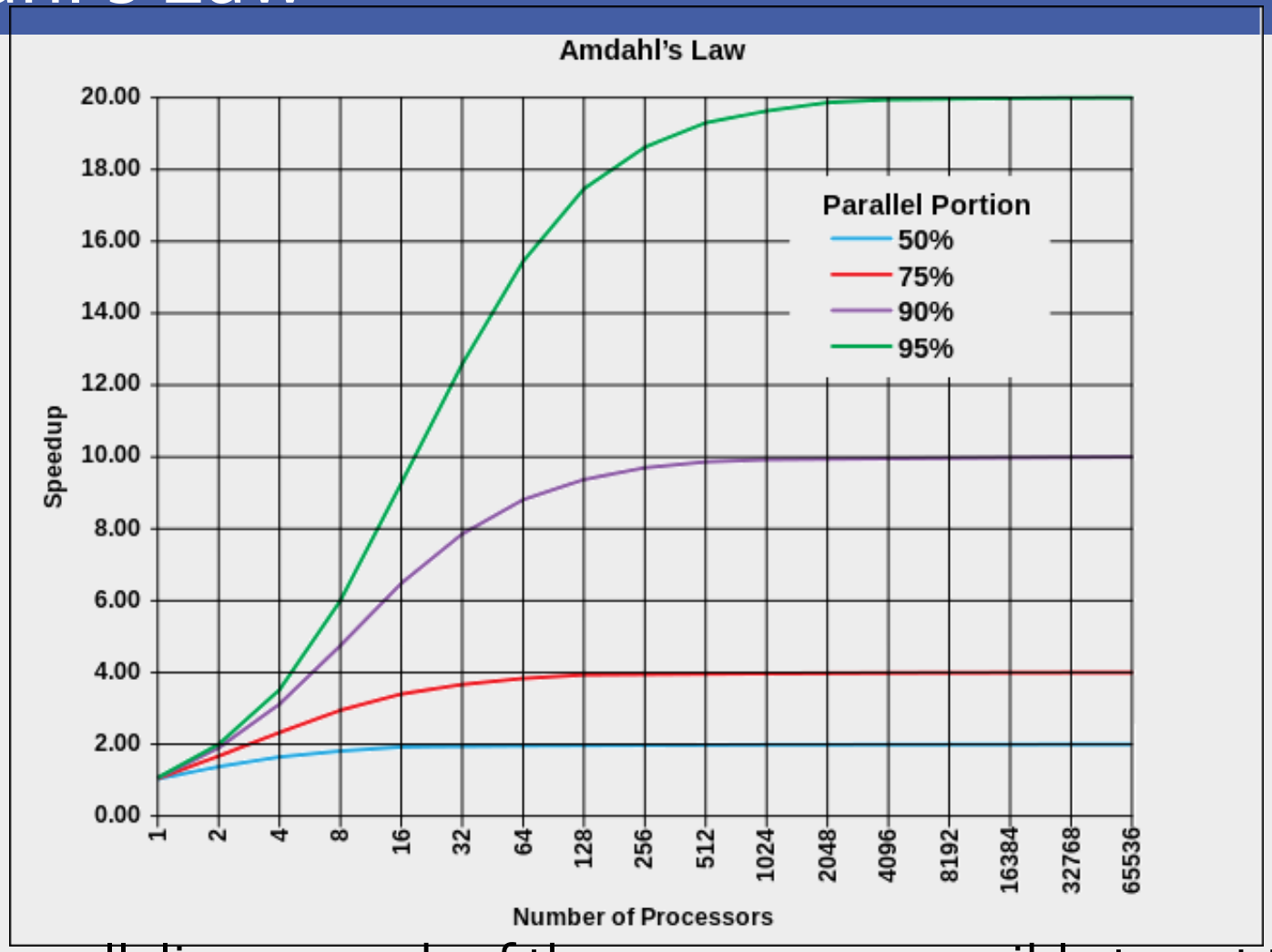
The computation time is divided in two parts



- the maximum speedup that can be obtained by using P processors is:

$$S_{\max} = \frac{1}{f + \frac{(1 - f)}{P}}$$

Amdahl's Law



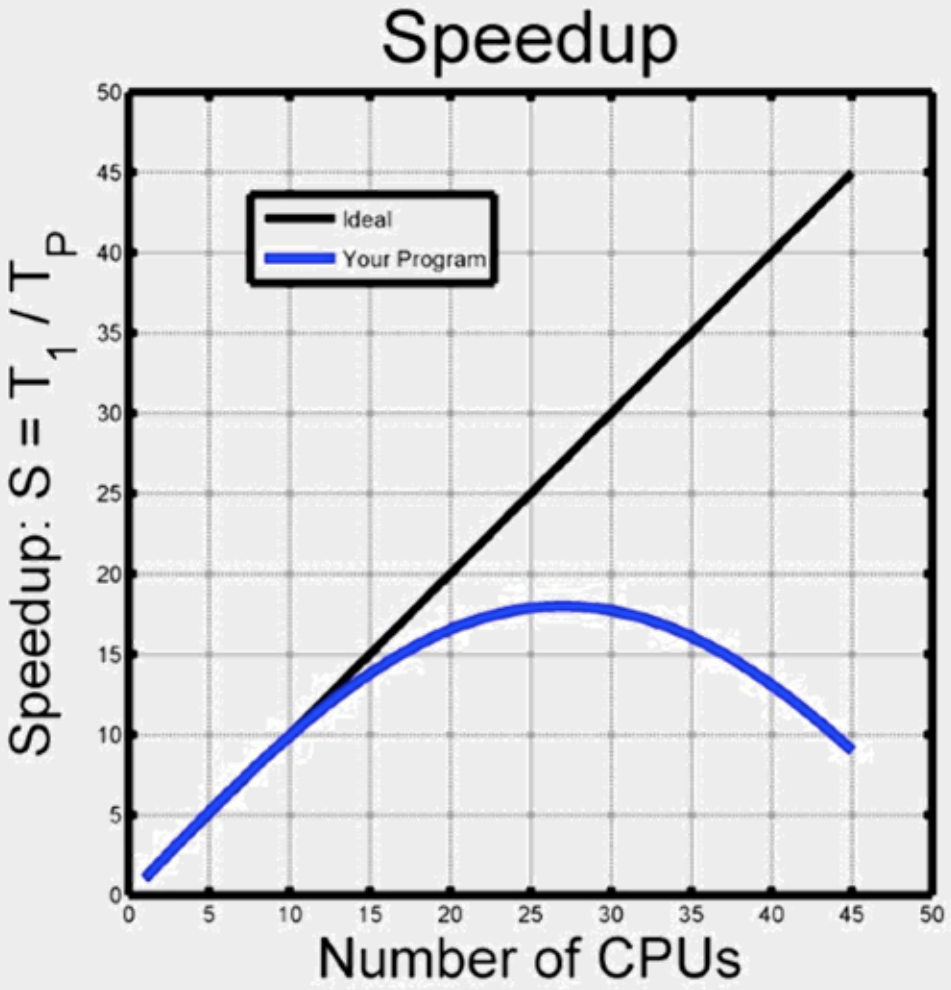
need to parallelize as much of the program as possible to get the best advantage from multiple processors

Back to Speedup

QUESTION: how many CPUs need to run this program?

- it's hard to tell how a program will behave on many processors
- **How to deal with it?**

DO a speedup test



And now…

Let's try to think parallel

Some Definitions We Need

PROCESS instance of a computer program that is being executed

THREAD a sequence of instructions within a process

RESOURCE any object of computer system that can be used by the process for its execution (processor, memory, programs, data, etc.)

Threads vs. Processes

PROCESSES

- typically independent
- carry considerably more state information
- have separate address spaces
- interact only through system-provided inter-process communication mechanisms

THREADS

- exist as subsets of a process
- share process state (memory and other resources)
- share their address space
- context switching between threads in the same process is typically faster than context switching between processes

Interprocess Communication

Process cooperation deals with three main issues

- Passing information between processes/threads
- Making sure that processes/threads do not interfere with each other
- Ensuring proper sequencing of dependent operations

- An ***independent*** process cannot affect or be affected by the execution of another process.

- A ***cooperating*** process can affect or be affected by the execution of another process

- Co-operating processes require an *interprocess communication* (IPC) mechanism that allow them to exchange data and information

The Consumer-Producer Problem

Suppose there are *producer* and *consumer* processes. Producer produces objects, which consumer uses for something. There is one Buffer object used to pass objects from producers to consumers.

The problem is to allow producers and consumers to access the Buffer while ensuring the following:

1. The shared Buffer should not be accessed by these processes simultaneously.
2. Consumers do not try to remove objects from Buffer when it is empty.
3. Producers do not try to add objects to the Buffer when it is full.

Producer & Consumer

Producer

```
while (1) {
        while (counter == BUFFER_SIZE)
              ; // do nothing
        //produce an item and put in nextProduced
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (1) {
        while (counter == 0)
              ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        //consume the item in nextConsumed
}
```


Updating of Shared Variable

Let' focus on **counter** variable.

- '**counter++**' & '**counter--**' could be implemented

counter ++	counter --
`reg1 = counter`	`reg2 = counter`
`reg1 = reg1 + 1`	`reg2 = reg2 - 1`
`counter = reg1`	`counter = reg2`

counter = ?

Possible Execution Interleaving

or

Producer Thread	Consumer Thread	reg1- value	reg2- value
`reg1 = counter`		5	
`reg1 = reg1 + 1`		6	
`counter = reg1`		`counter = 6`	
	`reg2 = counter`		6
	`reg2 = reg2 - 1`		5
	`counter = reg2`	**`counter = 5`**	

- Concurrent access to shared data may result in data inconsistency

Race Condition

The only way to deal with race conditions is through very careful coding.

Hence proccesses must be synchronized

The Critical-Section Problem

Critical section is a section of code or collection of operations in which only one process may be executing at a given time

- General structure of a typical process:

```
while (1){
        entry section
            critical section;
        exit section
            remainder section;
}
```

Critical section

The Critical-Section Problem

- The critical-section problem is to design a protocol that the processes can cooperate.
- Each process must request permission to enter its critical section.
- The section of code implementing this request is called entry section.
- The critical section may be followed by a section of code known as exit section.
- The remaining code is known as remainder section.

The Critical-Section Problem

Any solution to critical section problem must satisfy the following:

1. **Mutual Exclusion**:

   If a process P_i is executing in its critical section, then-no other processes can be executing in their critical sections.

2. **Progress**:

   The selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting**:

   There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Types of solutions to CS problem

- Software-based
  - Some known algorithms (as Peterson, Bakery, Busy Waiting, and others)
- Hardware-based
  - locks, test_and_set Instruction, semaphores, monitors…

Semaphores

Semaphore S is integer variable used by processes to send signals to other processes

- Less complicated
- Accessed only through two standard atomic (indivisible) operations:
  - *P()* or *wait()*
  - *V()* or *signal()*

```
Wait(S) {
      while (S <= 0)
            ; // do
nothing
      S--;  }
```

```
Signal(S) {
      S++;

}
```


Semaphores

- ***Counting semaphore*** – integer value can range over an unrestricted domain
- ***Binary semaphore*** – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as ***mutex locks***
  - Provides mutual exclusion

```
Semaphore mutex; // initialized to 1  do
{     wait (mutex);          // Critical
Section    signal (mutex);              //
remainder section   } while (TRUE);
```


Synchronization problems

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

  Let *S* and *Q* be two semaphores initialized to 1

P_0
wait (S);
wait(Q);
…
signal(S);
signal(Q);

P_1
wait (Q);
wait(S);
…
signal(Q);
signal(Q);

Synchronization problems

- **Starvation(=indefinite blocking)**
  - Related to deadlocks (but different)
  - Occurs when a process waits indefinitely in the semaphore queue
  - For example, assume a LIFO semaphore queue
    - Process pushed first into it might not get a chance to execute
- **Priority Inversion**
  - A situation when a higher-priority process needs to wait for a lower-priority process that holds a lock

Classical Problems of Synchronization

Classical problems used to test newly-proposed synchronization schemes

- Bounded-Buffer Problem (Producer-Consumer Problem)
- Readers and Writers Problem
- Dining-Philosophers Problem
- Sleeping-Barber Problem

Solution of the Bounded-Buffer Problem

- **Shared data structure**

```
int n;
semaphore mutex = 1; //Guards the access to the
                        buffer pool
semaphore empty = n; // Counts the number of
                      empty buffers
semaphore full = 0; // Counts the number of full
                       buffers
```

Solution of the Bounded-Buffer Problem

- **The structure of the producer process**

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty); //suspend self if the buffer is full
    wait(mutex); //get access to the buffer
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex); //release the lock to the buffer
    signal(full); //
}
```


Solution of the Bounded-Buffer Problem

- **The structure of the consumer process**

```
while (true) {
        wait(full); //suspend self if the buffer is empty
        wait(mutex); // get access to the buffer
                ...
                /* remove an item from buffer to
                        next_consumed */
                ...
        signal(mutex); // release the lock to the buffer
        signal(empty);
                ...
                /*consume the item in next_consumed */
                ...
}
```

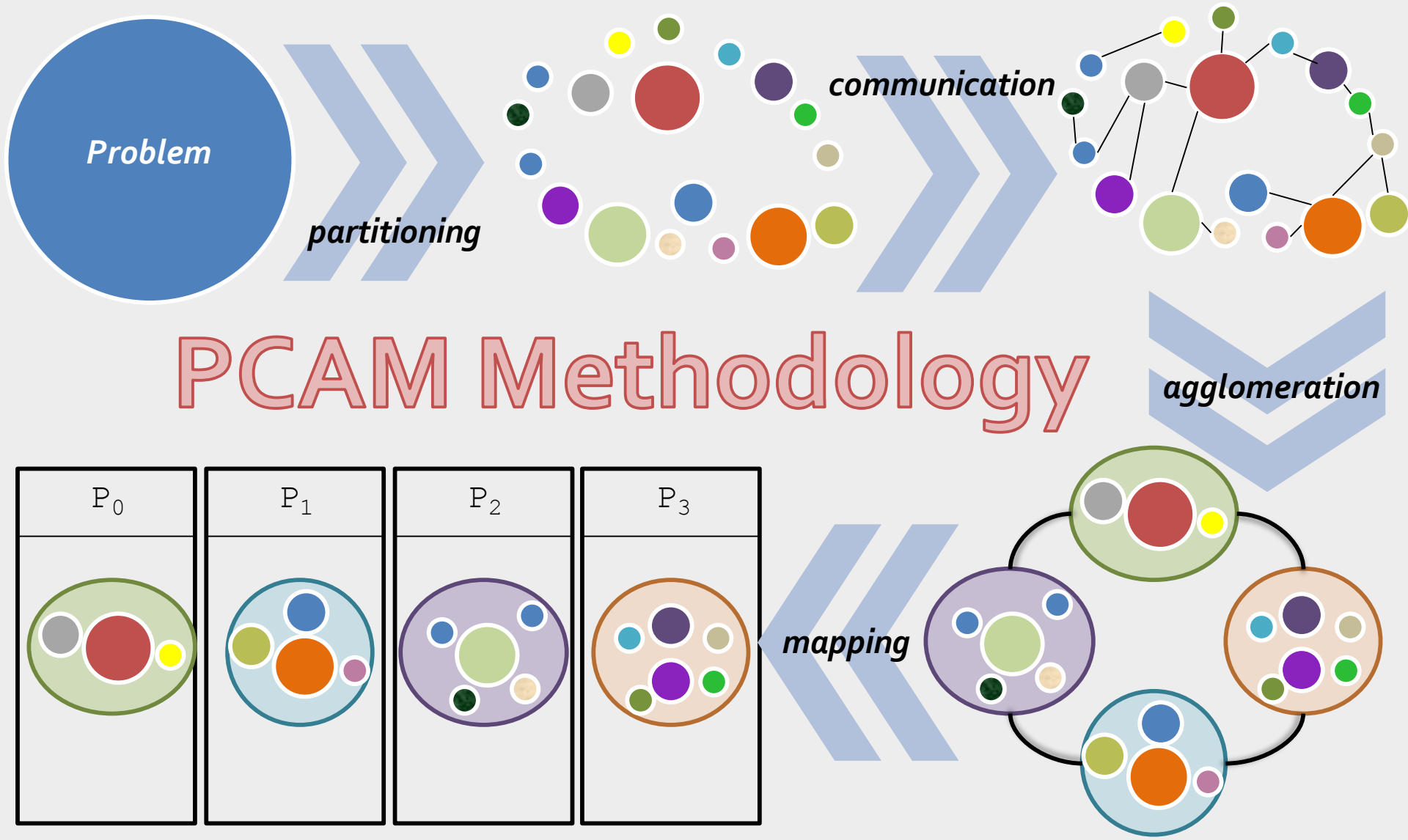

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - *Readers* – only read the data set; they do not perform any updates
  - *Writers* – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- **Solution: try find it by yourself**

Designing Parallel Programs

PhD Katerina Bolgova
eScience Research Institute & HPC Department

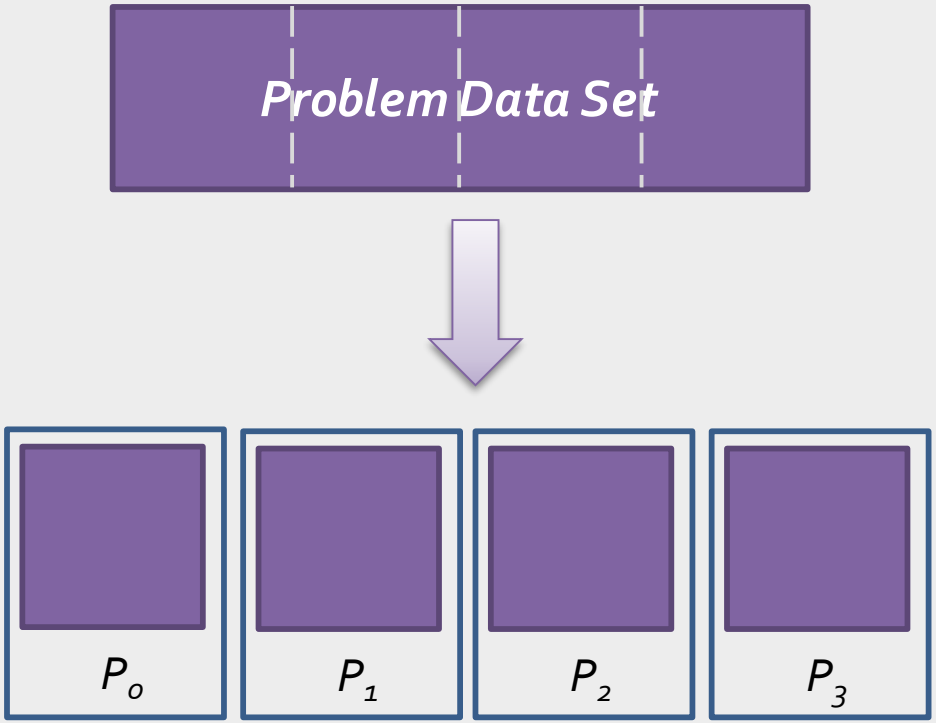
Designing Parallel Programs



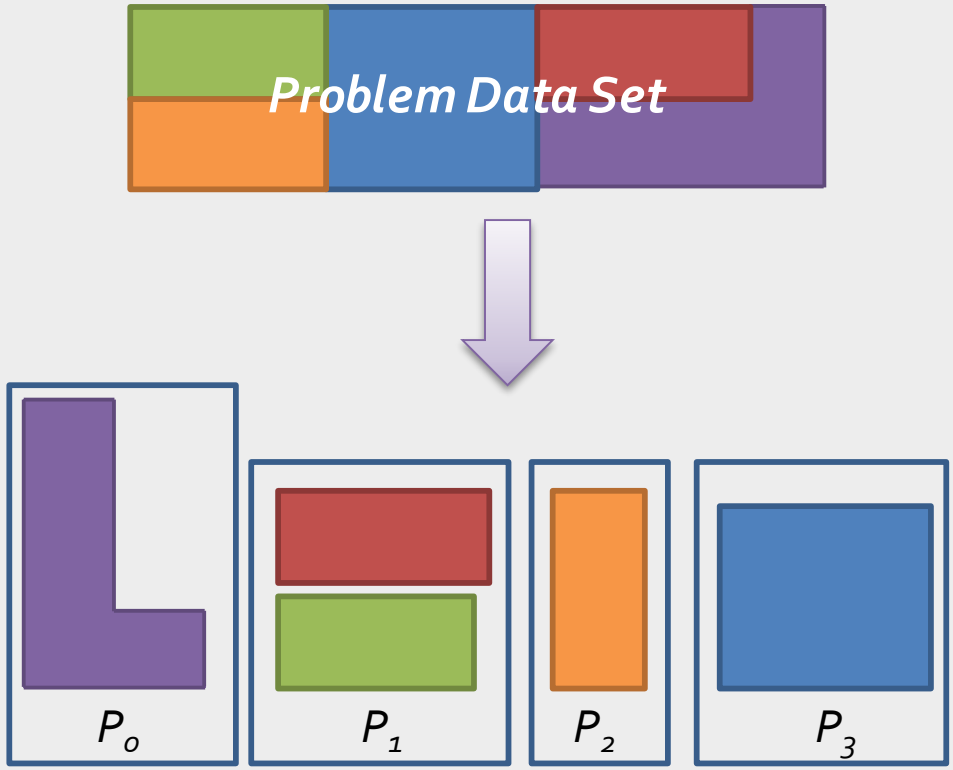
Designing Parallel Programs

Partitioning

Domain Decomposition



Functional Decomposition



Partitioning Design Checklist

1. Does your partition define at least an order of magnitude more tasks than there are processors in your target computer?

2. Does your partition avoid redundant computation and storage requirements?

3. Are tasks of comparable size?

4. Does the number of tasks scale with problem size?

5. Have you identified several alternative partitions?

Designing PP. Communication

- The need for communications between tasks depends on your problem:
  - *Most* parallel applications do require tasks to share data with each other
  - *Embarrassingly parallel problems* tend to require little or no communication of results between tasks

- **Some factors to consider:**
  - Cost of communications
  - Latency and Bandwidth
  - Visibility of communications
  - Synchronous and asynchronous communications
  - …

Communication Design Checklist

1. Do all tasks perform about the same number of communication operations?

2. Does each task communicate only with a small number of neighbors?

3. Are communication operations able to proceed concurrently?

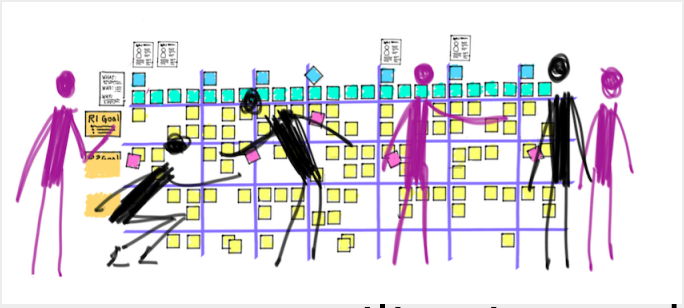
4. Is the computation associated with different tasks able to proceed concurrently?

Designing Parallel Programs

- ***Agglomeration***

  - Tasks are combined into larger tasks to improve performance or to reduce development costs (if necessary).

- ***Mapping***



  - Main goals: *maximizing* processor utilization and *minimizing* communication costs.
  - Mapping can be specified statically or determined at runtime by load-balancing algorithms.