

# Foundations of High Performance Computing

Lecture 04: Parallel concepts  
and performance evaluations



**High Performance Computing course**

2023-2024 Stefano Cozzini

# Agenda

Parallel programming paradigm

Parallel programming concepts

Parallel performance

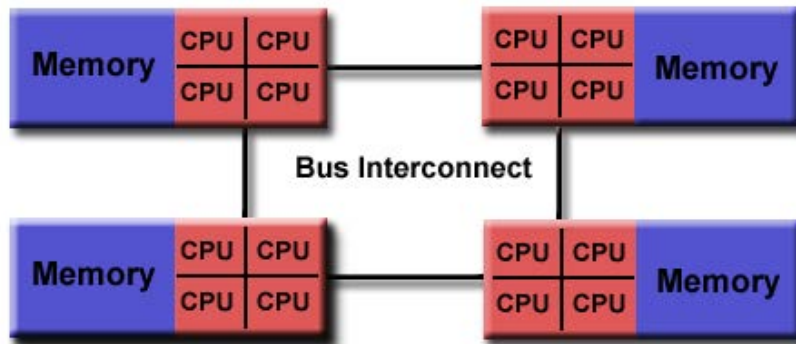
Ahmdal /Gustafson law

# 2 main parallel paradigms

## DIDACTED BY MEMORY ORGANIZATION

### shared memory

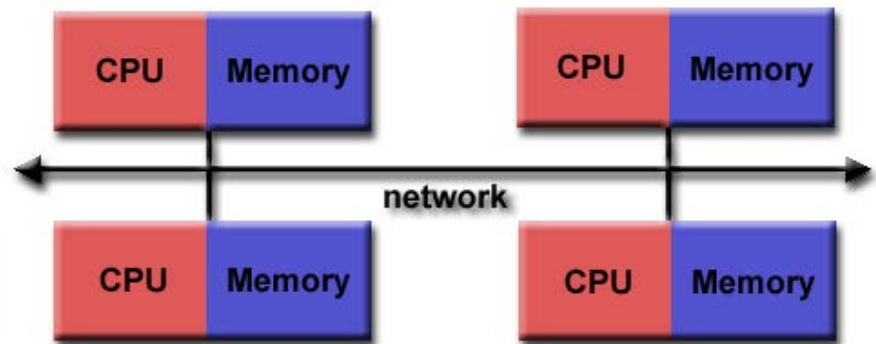
Single memory view, all processes (usually threads) could directly access the whole memory



### distributed memory

#### Message Passing

all processes could directly access only their local memory.



# Pro&Cons

## • Pros

- Unique 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

## • Cons

- Cannot scale to large number of cores
- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.
- Non uniform memory access time on modern CPU architecture

## • Pros

- Memory is scalable with the number of processors. Increase the number of processors and the size of memory increases proportionately.

## • Cons

- Data is scattered on separated address spaces
- The programmer is responsible for many of the details associated with data communication between processors.
- Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.

# Programming enviroment

## • Shared

- Ad hoc compilers
- Source code directives (trivial portability)
- Standard unix shell to run the program
- Standard: OpenMP

## • Distributed

- Standard Compilers
- Communication libraries ( not so trivial portability\_
- Ad hoc command to run the program
- Standard MPI

# Shared memory approach: a first basic example

loop parallelization with OpenMP

```
#pragma omp parallel for  
for(int i=0; i<n; ++i)  
    c[i]= a[i]+b[i];
```

Compile with correct flag: -f openmp

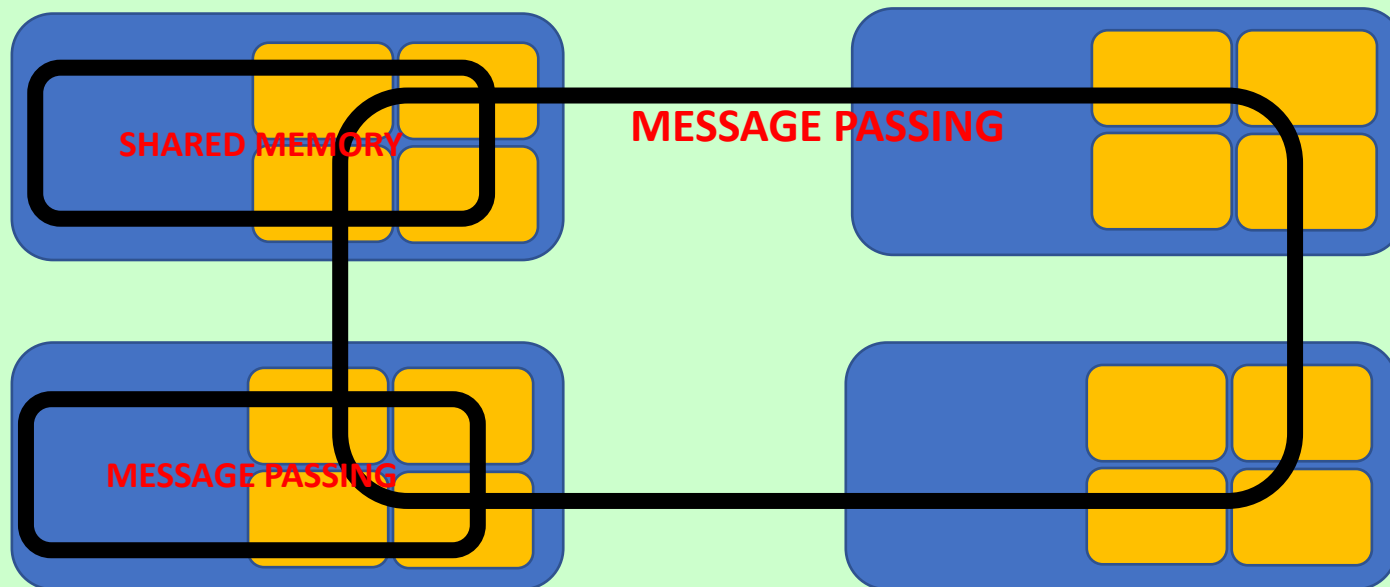
```
gcc -fopenmp mycode.c
```

# Message Passing approach

- Using the de-facto standard : MPI message passing interface
  - A standard which defines how to send/receive message from a different processes
- Many different implementation
  - OpenMPI
  - Intel-MPI
- They all provide a library which provide all communication routines
- To compile your code you have to link against a library
- Generally a wrapper is provided (mpif90/mpicc)

# HPC Architecture vs Paradigms

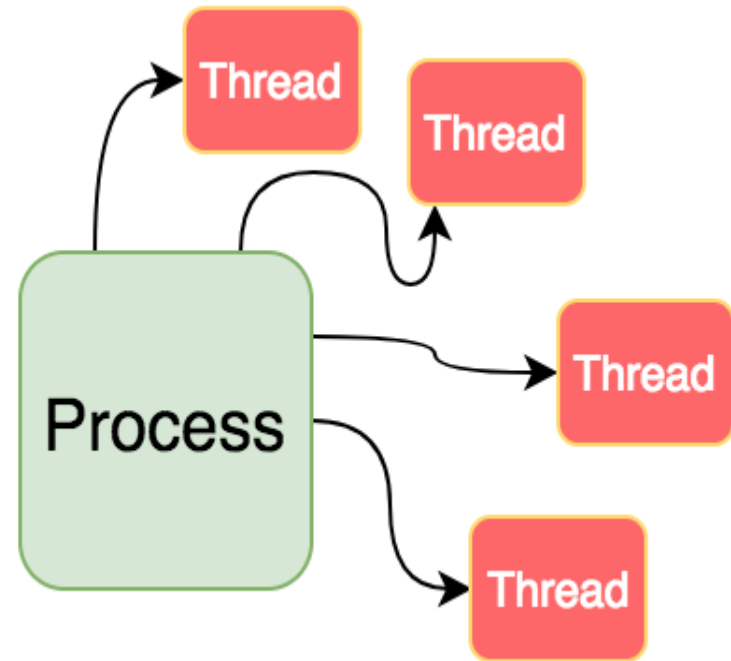
**A Cluster of Shared Memory Nodes:  
a distributed memory machine**





# Important note

- It is trivial to implement MP approach on Shared Memory machine..
  - Each Linux process has its own private memory
- It is impossible to implement shared memory approach on distribute memory machine..
  - Threads are spawned by a single linux process and so they share the same memory



# Other paradigm available

- Mixed/hybrid approach..
  - MPI + OpenMP
- Specific SDK for specific devices
  - CUDA for Nvidia GPU
- Write once run everywhere:
  - OpenCL
  - OpenACC:
    - OpenACC is about giving programmers a set of tools to port their codes to new heterogeneous system without having to rewrite the codes in proprietary languages.

# Agenda

Parallel programming paradigm



Parallel programming concepts

Parallel performance

Ahmdal /Gustafson law

# Why parallelism ?

For two main reasons:

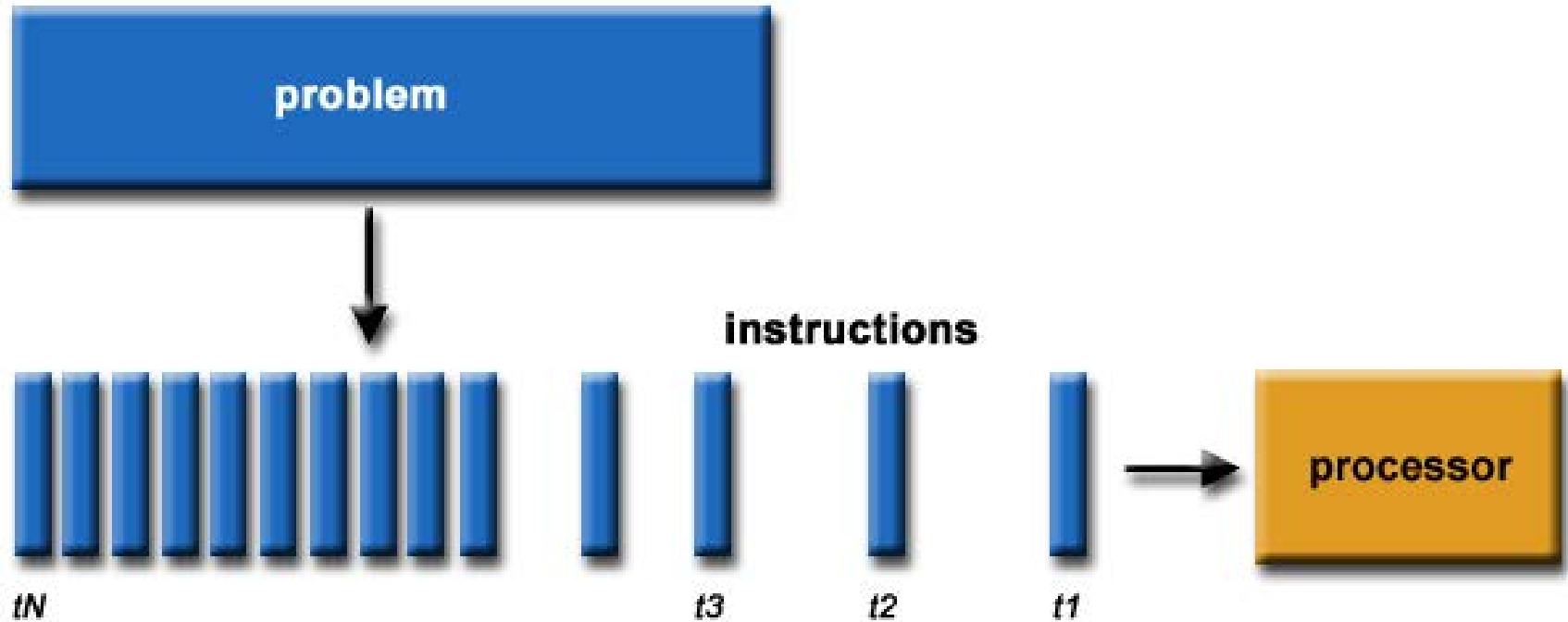
## 1. Time-to-solution

- To solve the same problem in a **smaller time**
- To solve a **larger problem** in the same time

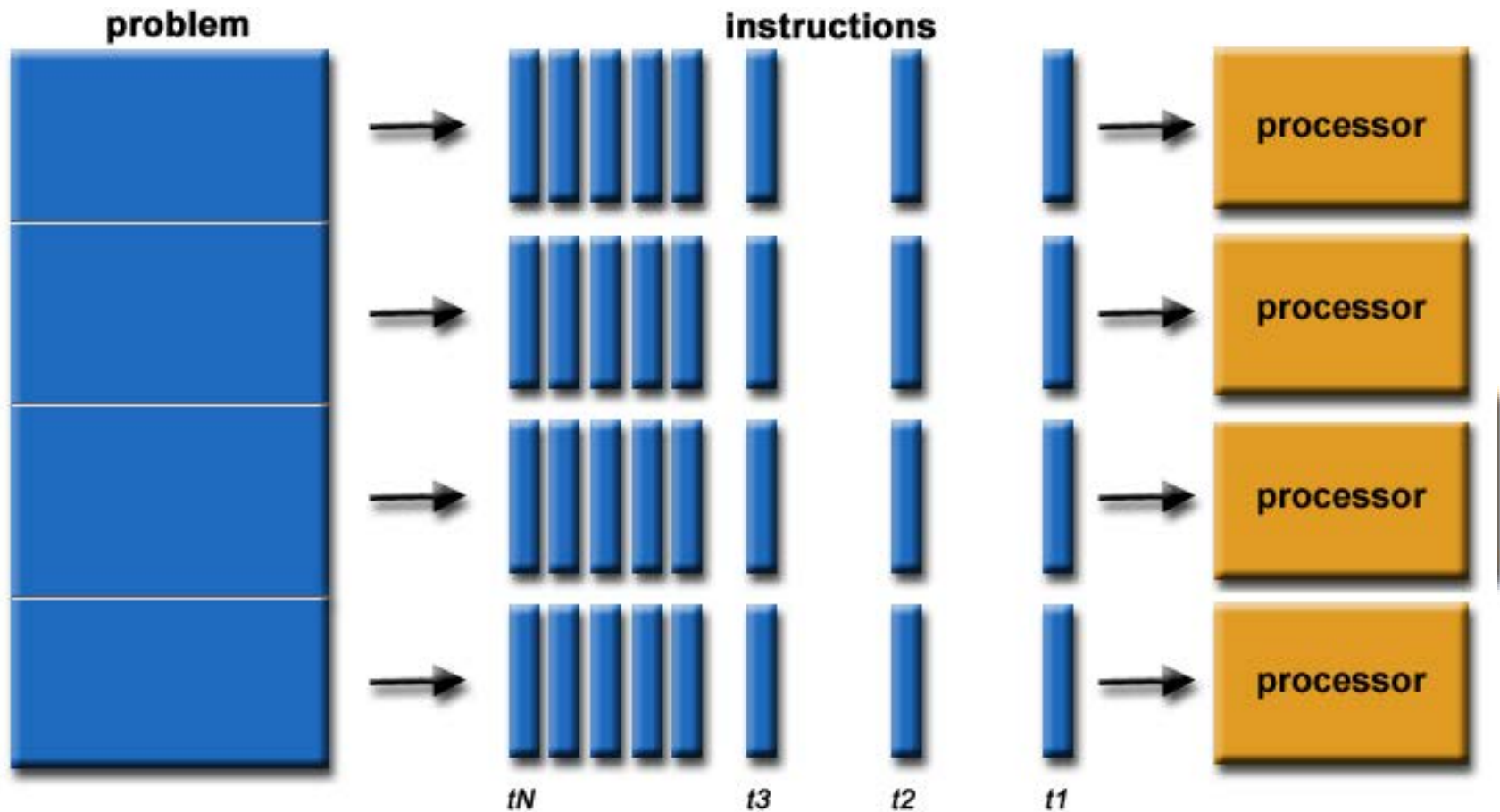
## 2. Problem size ( ~ data size)

- To solve a problem that could *not* fit in the memory addressable by a single computational units (or that could fit in the space around a single computational units without serious performance loss)

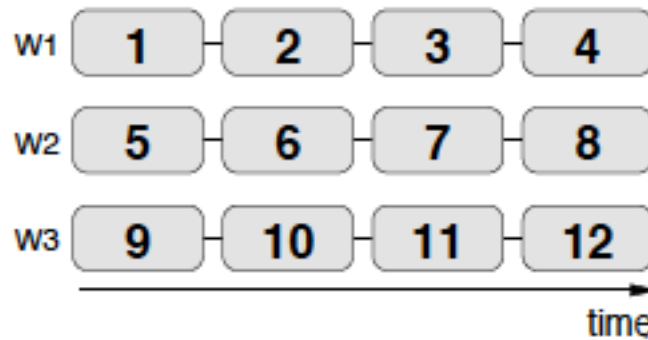
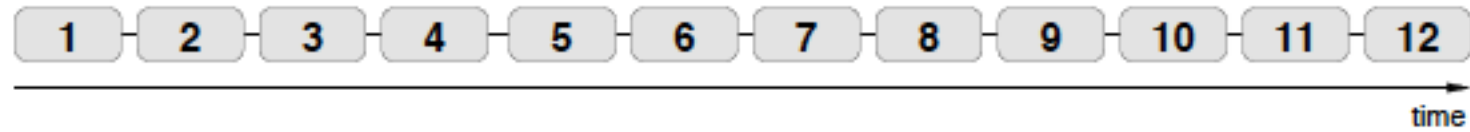
# Serial execution



# Parallel execution

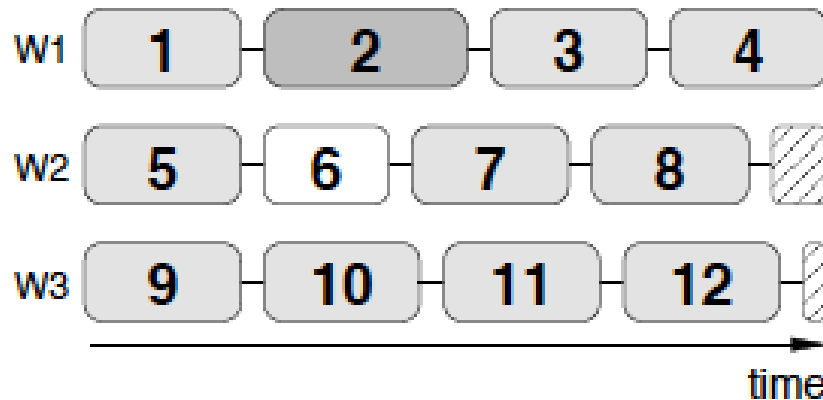


# Running in parallel



- Execution time reduces from 12 secs to 4 secs!

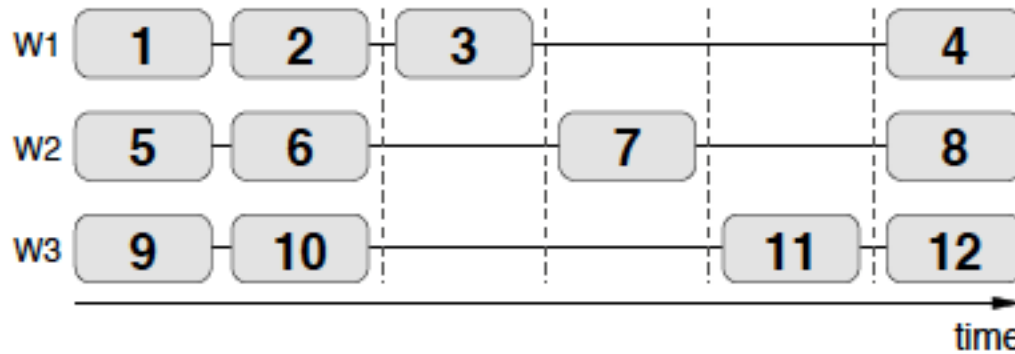
# Load imbalance..



- What if all processors can't execute tasks with the same speed?
- Load imbalance (ending parts for W2 and W3)



# Dependency among tasks



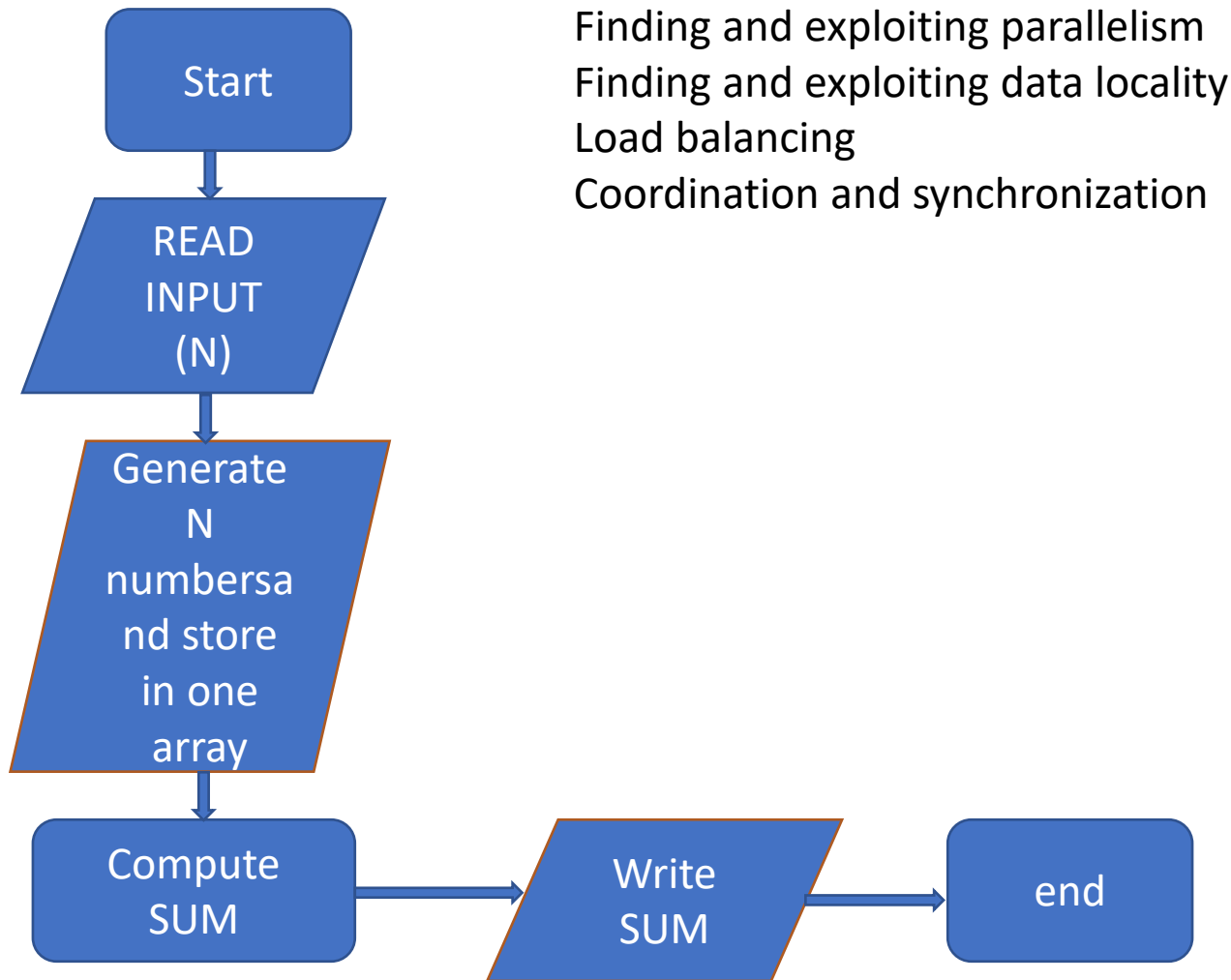
- What if section 11 depends on section 7 that depends on section 3 ?
- Time increase from 4 to 6 !

# Principle of parallel computing

- Finding and exploiting parallelism
- Finding and exploiting data locality
- Load balancing
- Coordination and synchronization
- Parallel performance
  - Speedup, efficiency
  - Ahmdal Law/Gustafson Law
  - Performance modeling

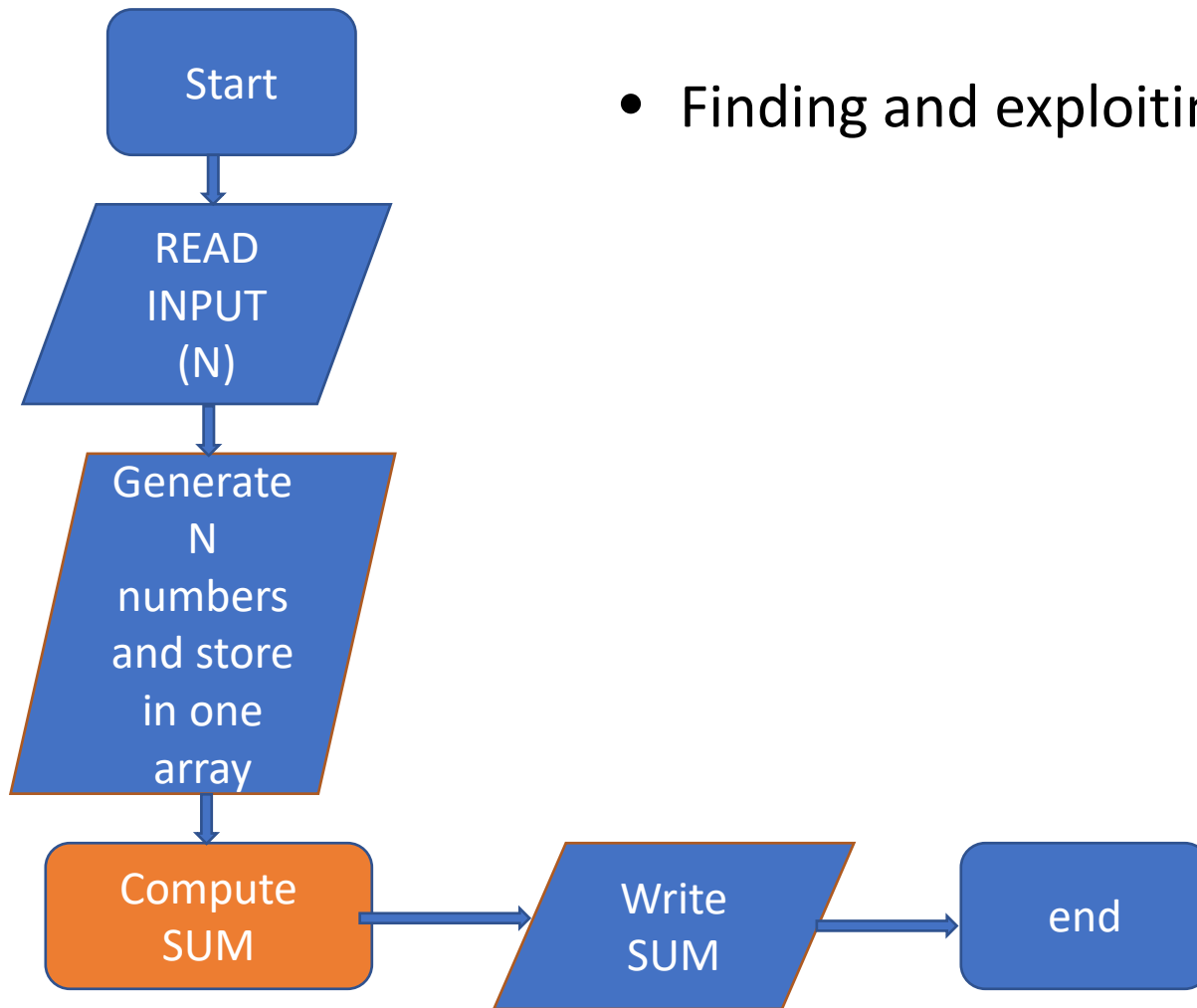
All of these things make parallel programming more difficult than sequential programming.

# The simplest algorithm: sum of N numbers

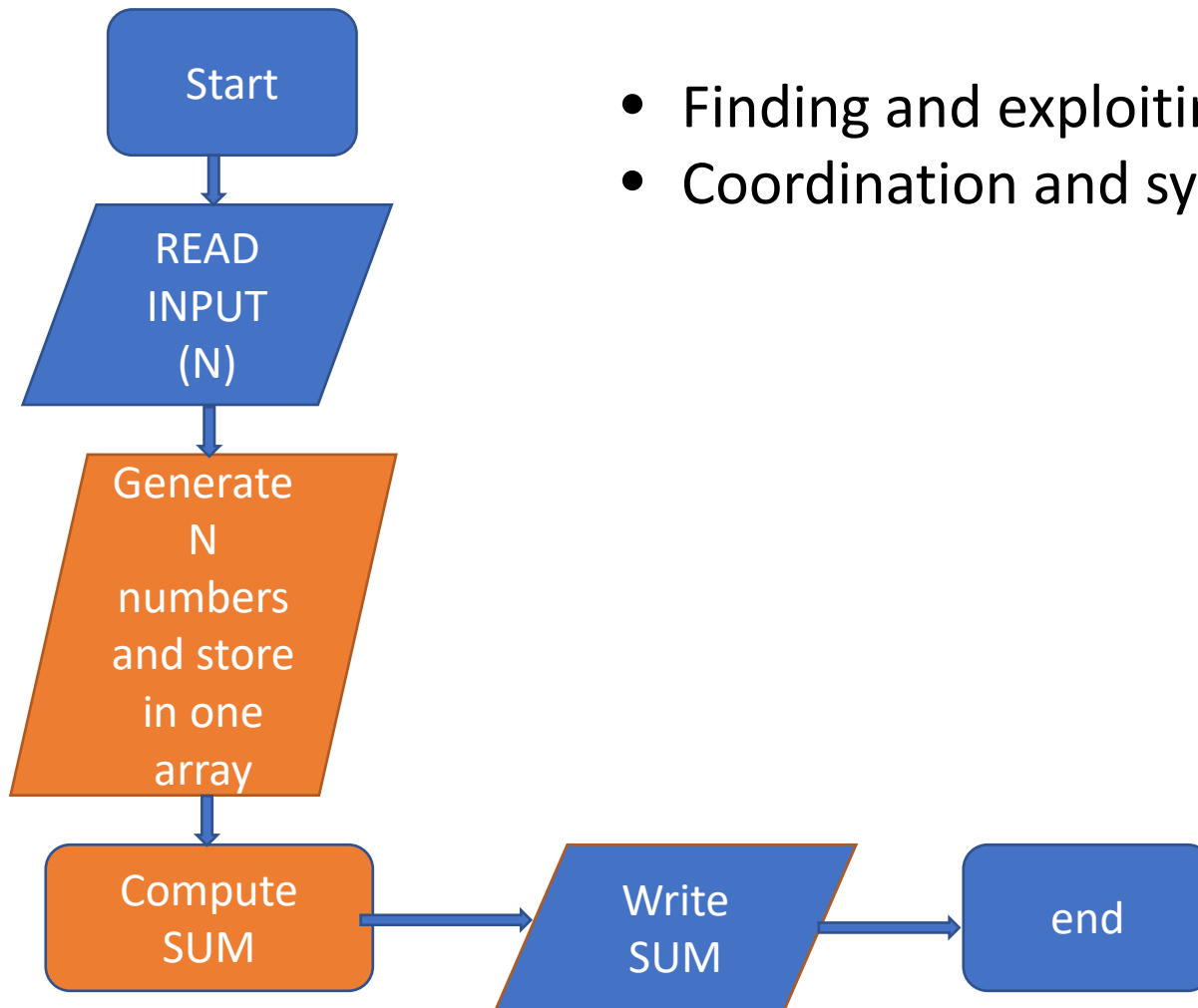


# The simplest algorithm: sum of N numbers

- Finding and exploiting parallelism



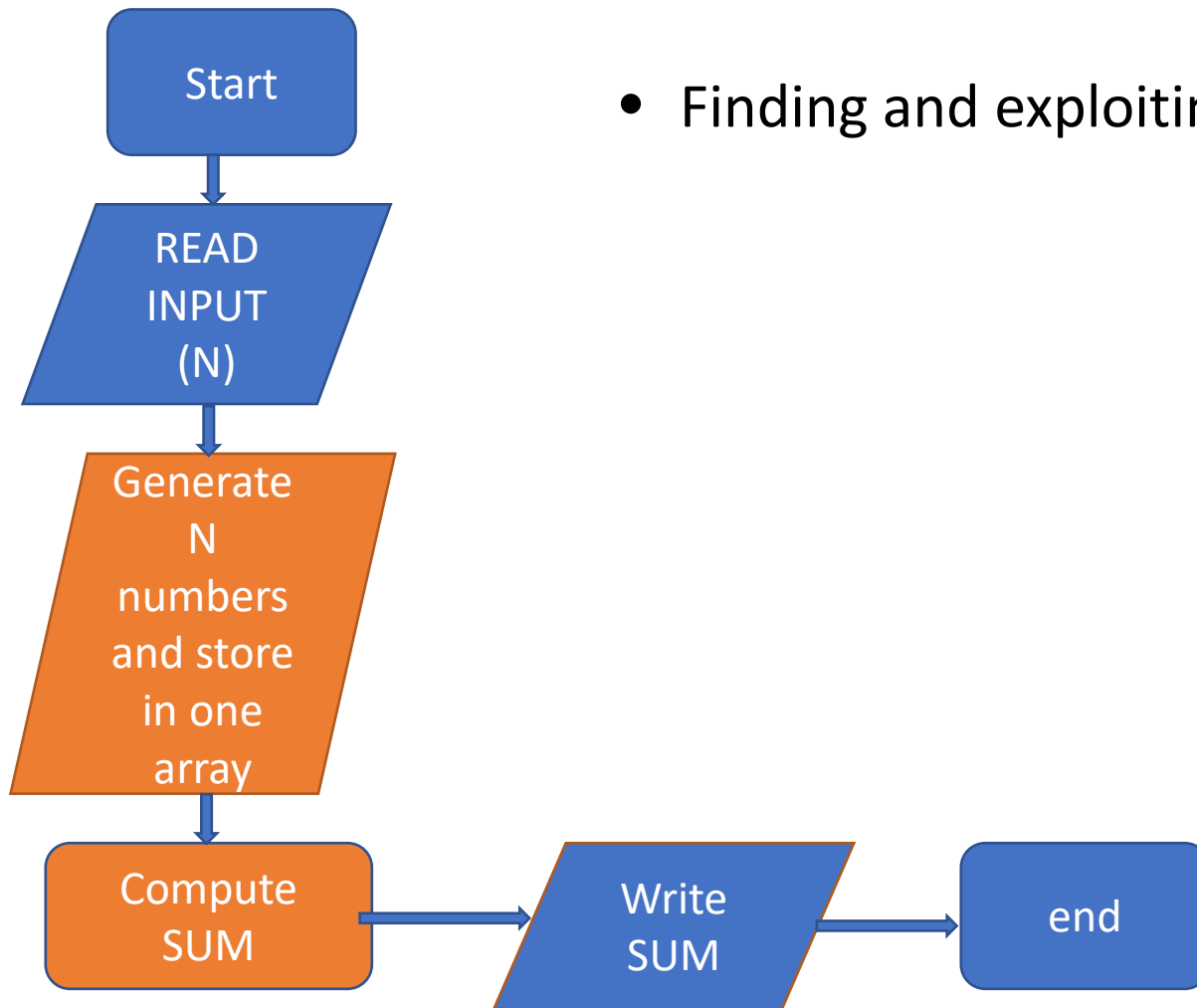
# The simplest algorithm: sum of N numbers



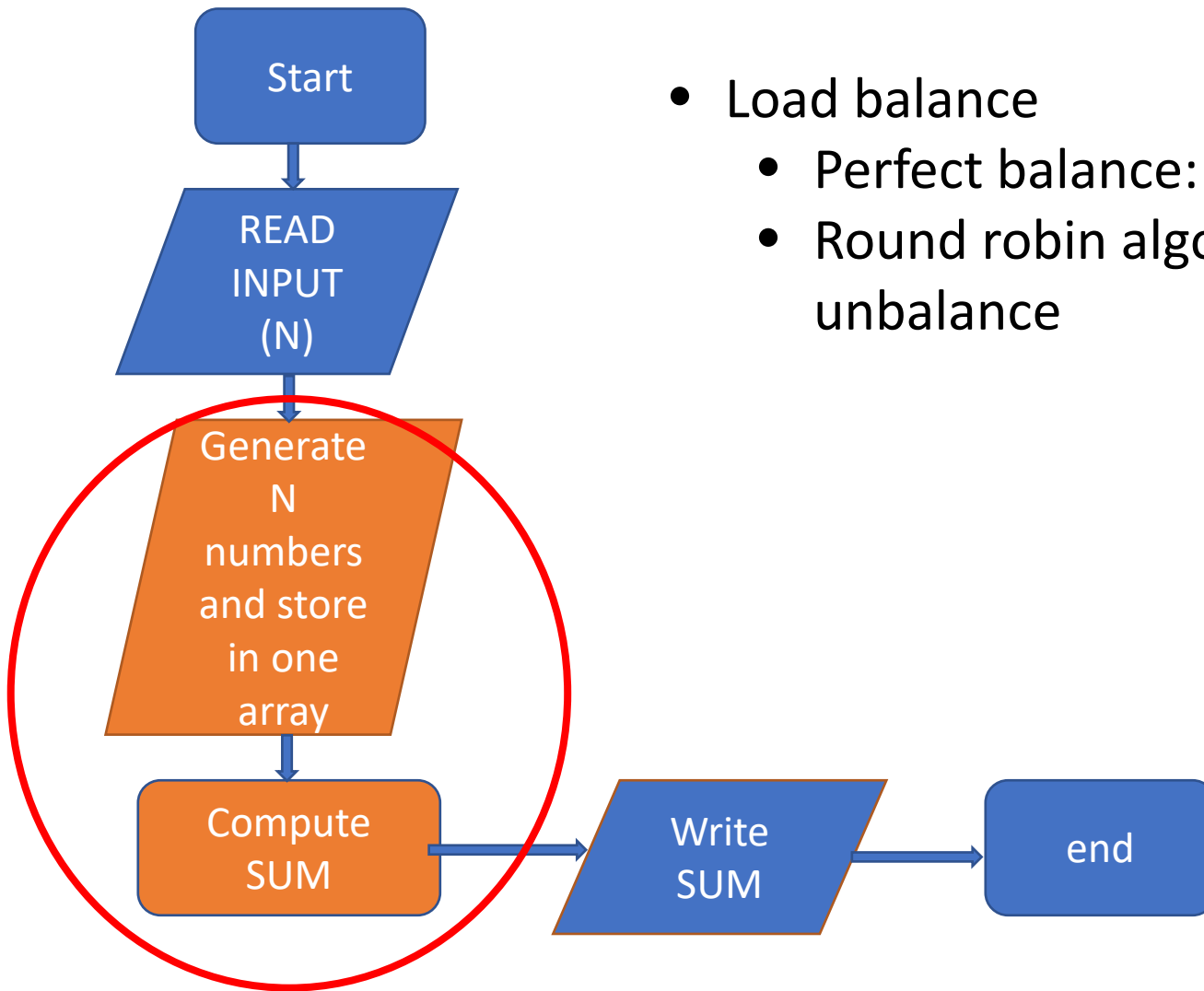
- Finding and exploiting data locality
- Coordination and synchronization

# The simplest algorithm: sum of N numbers

- Finding and exploiting data locality

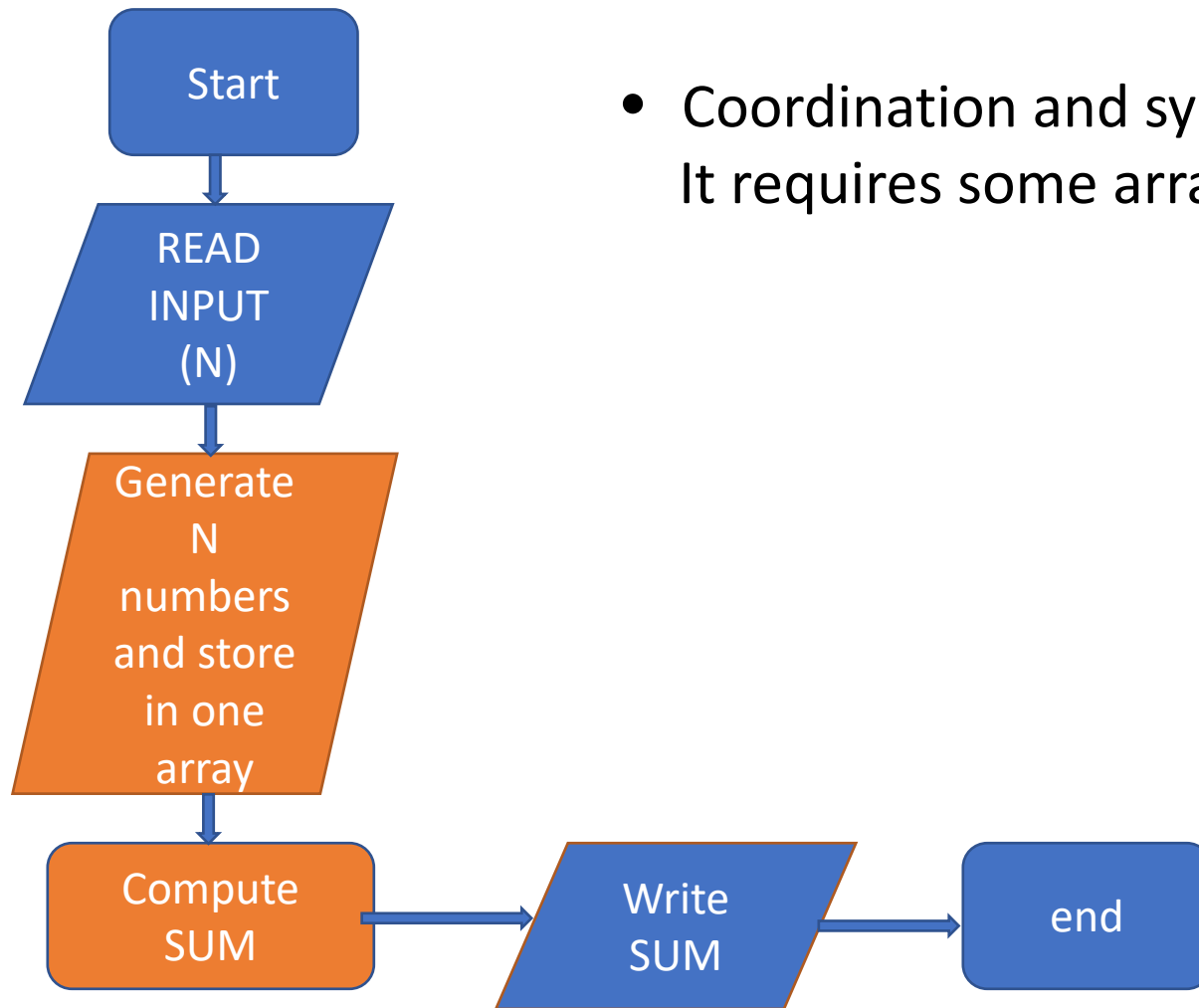


# The simplest algorithm: sum of N numbers



- Load balance
  - Perfect balance:  $[N/P]=0$
  - Round robin algorithms minimize unbalance

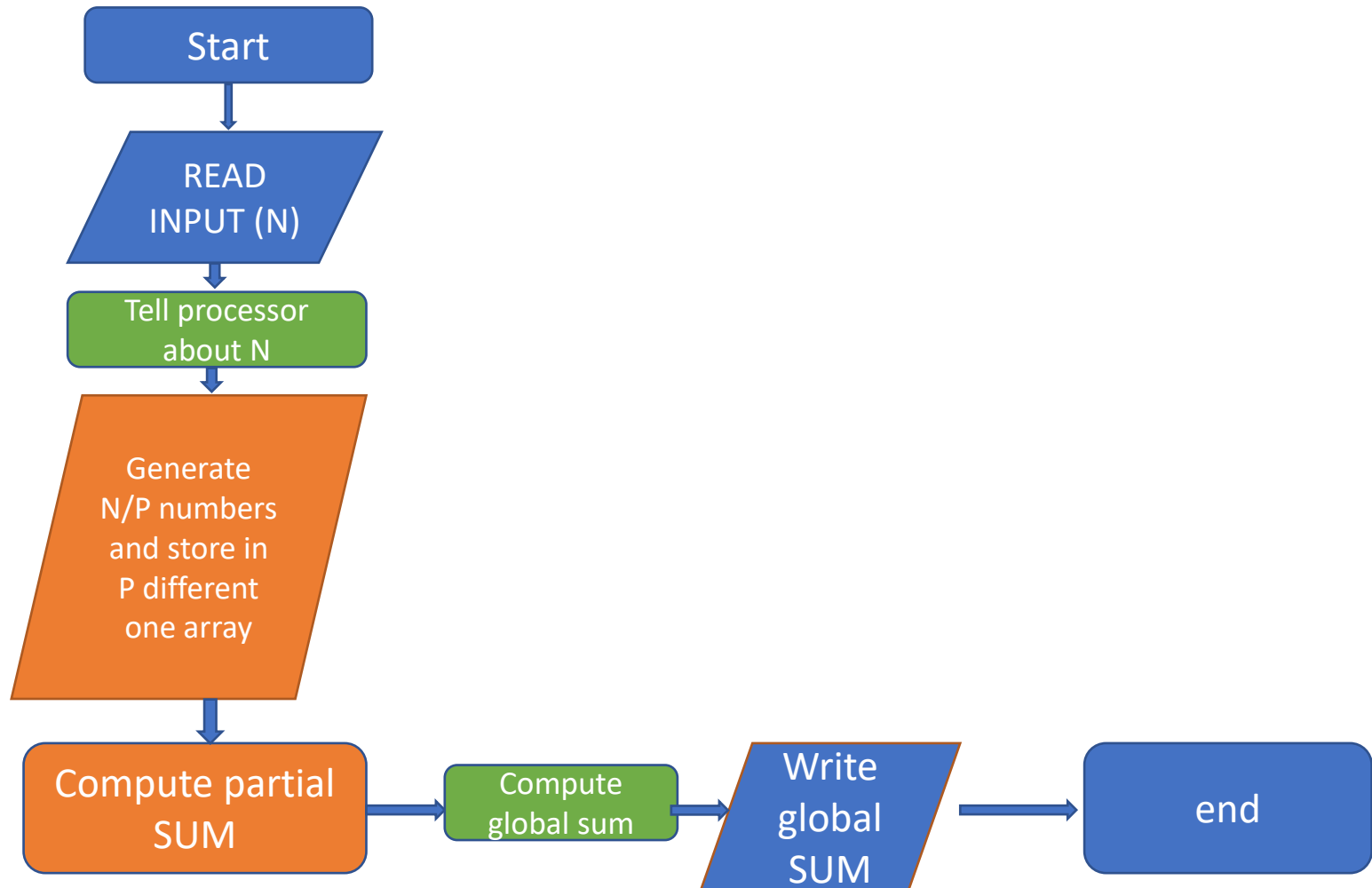
# The simplest algorithm: sum of N numbers



- Coordination and synchronization  
It requires some arrangements ...



# The simplest algorithm: sum of N numbers



# Agenda

Parallel programming paradigm



Parallel programming concepts

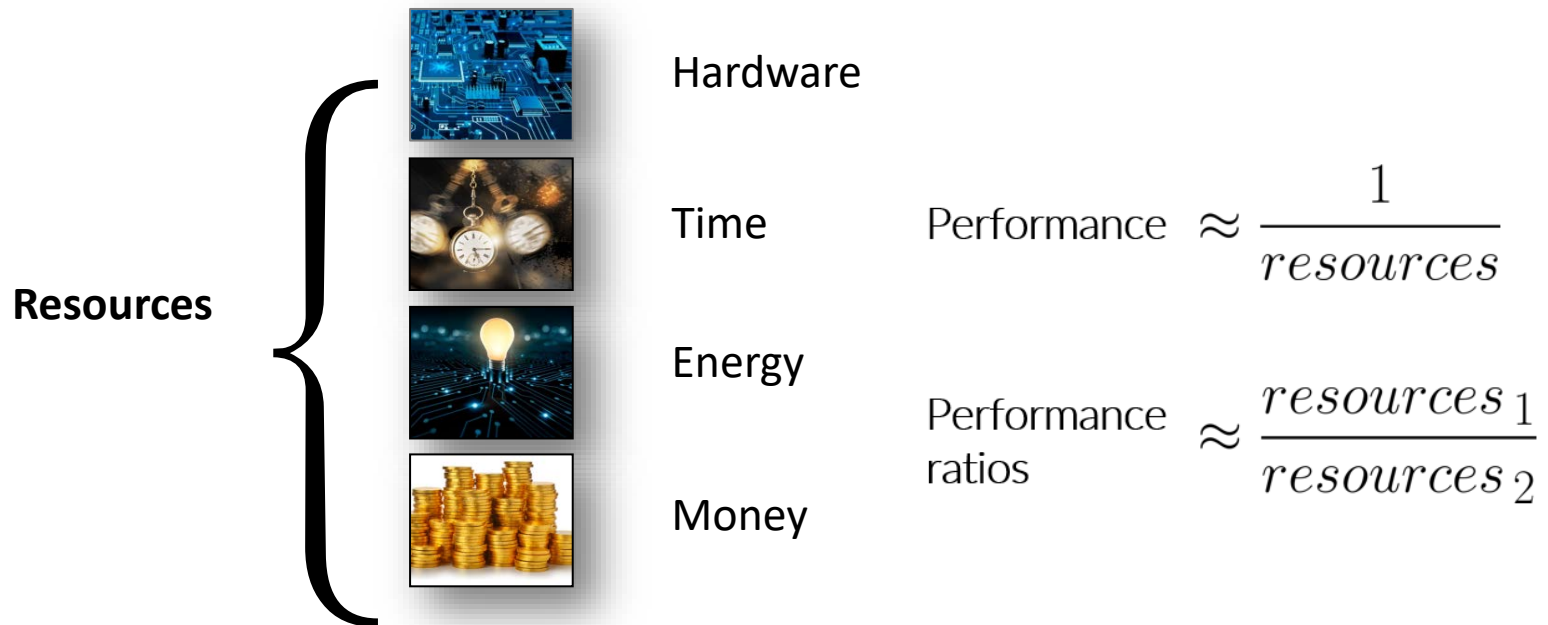


Parallel performance

Ahmdal /Gustafson law

# What is parallel performance ?

- «performance» is a tag that can stand for many things.
- Here «performance» means the relation between the computational requirements and the computational resources needed to meet those requirements.



# Scaling...

- Scaling or scalability: some sort of ratio between the performance and the “size” of the HPC infrastructure
- Usual way to measure size: # of processors
  - The ability for some application to increase speed when the size of the HPC is increased
  - The ability for some application to solve larger problems when the size of the HPC increases..

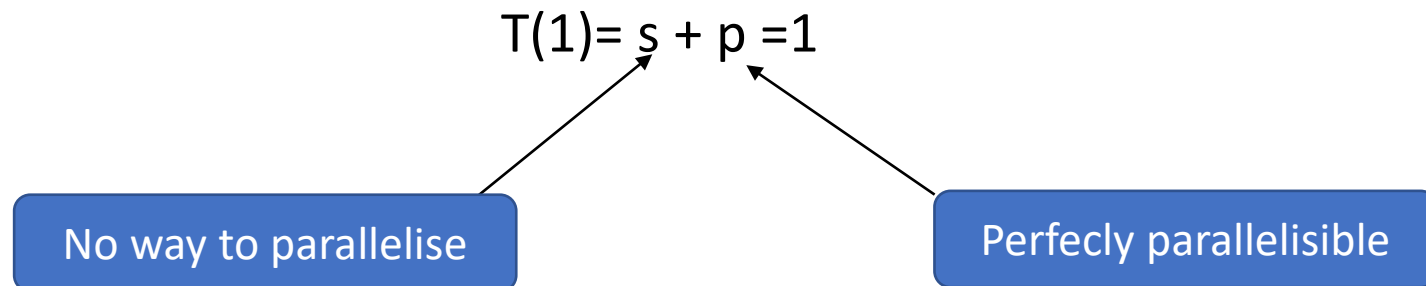
# Some more specific questions on scalability

- How much faster can a given problem be solved with  $N$  workers instead of one?
- How much more work can be done with  $N$  workers instead of one?
- What impact for the communication requirements of the parallel application have on performance?
- What fraction of the resources is actually used productively for solving the problem?

Identify basic limitations of code implementations or algorithms for parallel processing

# Assumptions

- Underlying hardware is perfectly scalable
- Basic workload may have pure serial and pure parallel contributions
- P „workers“ have to perform either
  - Fixed amount of work as fast as possible Amdahl's law
  - Increasing amount of work ( $\sim P$ ) in constant time Gustfson's law
- Time based view:
  - Time to execute the serial( $P=1$ ) workload on one worker:  $T(1)=1$
  - Basic assumption(serial/parallel workload):



# Definitions

$n$	Problem size
$T_s(n)$	Serial run-time for a problem of size $n$
$T_p(n)$	Parallel run-time with $p$ processes for a problem of size $n$
$p$	Number of computing units
$f_n$	Intrinsic sequential fraction of the problem of size $n$
$k(n, t)$	Parallel overhead

# Speed-up and efficiency

- $T(P)$  is the time to execute „some workload“ with  $P$  workers
- **Parallel Speed-Up:** How much faster do I execute the given workload on  $P$  workers?

Parallel Speed-Up:  $Sp(n, t) = \frac{T_s(n)}{T_p(n)}$

- **Efficiency:** How efficient do I use the workers in average?

Parallel Efficiency:  $Eff(n, t) = \frac{T_s(n)}{p \times T_p(n)} = \frac{Sp(n, t)}{p}$

- Warning: These metrics are relative to the time (performance) of a single worker → These metrics are not performance metrics!



# Some observations

- If  $\text{Speedup}(p) = p$  we have perfect speedup (also called linear scaling)
  - For perfect speedup Efficiency  $(p) = 1$
  - Ideal case: holy grail for all HPC users..
- speedup compares an application with itself on one and on  $p$  processors
  - Sometimes more useful to compare:  
The execution time of the best serial application on 1 processor against the execution time of best parallel algorithm on  $p$  processors

Understanding why an application is not scaling linearly will help finding ways improving the applications performance on parallel computers.

# Superlinear speed-up

- Question: can we find “*superlinear*” speedup, that is

$$\text{Speedup}(p) > p ?$$

Choosing a bad “baseline” for  $T(1)$

- Old serial code has not been updated with optimizations
- Parallel code on one processor does much more work

Shrinking the problem size per processor

- May allow it to fit in small fast memory (cache)

# Agenda

Parallel programming paradigm



Parallel programming concepts



Parallel performance



Ahmdal /Gustafson law

# Ahmdal's law

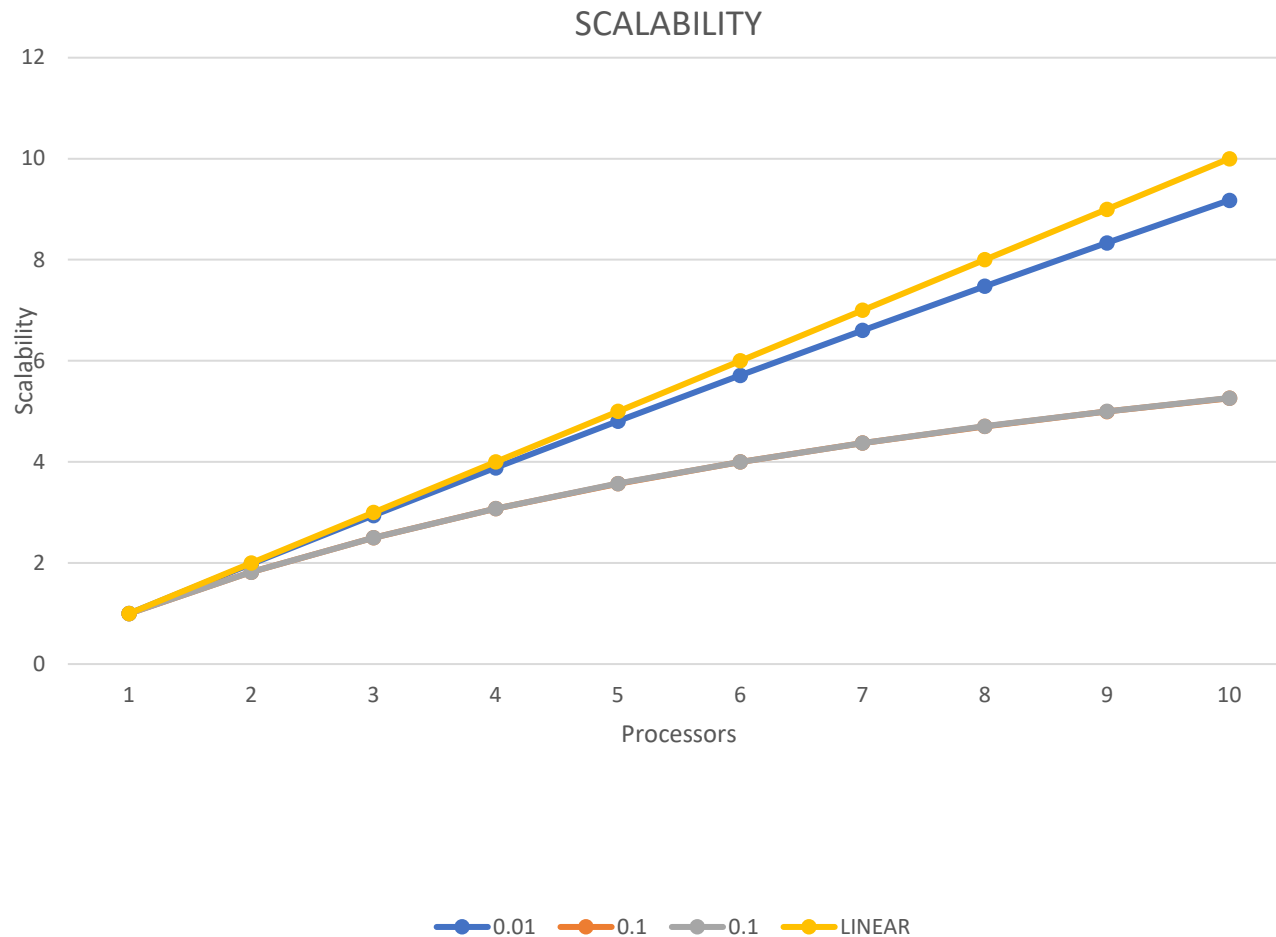
- $S(P) = T(1)/T(P)$
- $T(1) = s + p = 1$
- $T(P) = s + p/P$
- After a little bit of basic math:

$$S(P) = 1 / (s + (p/N))$$

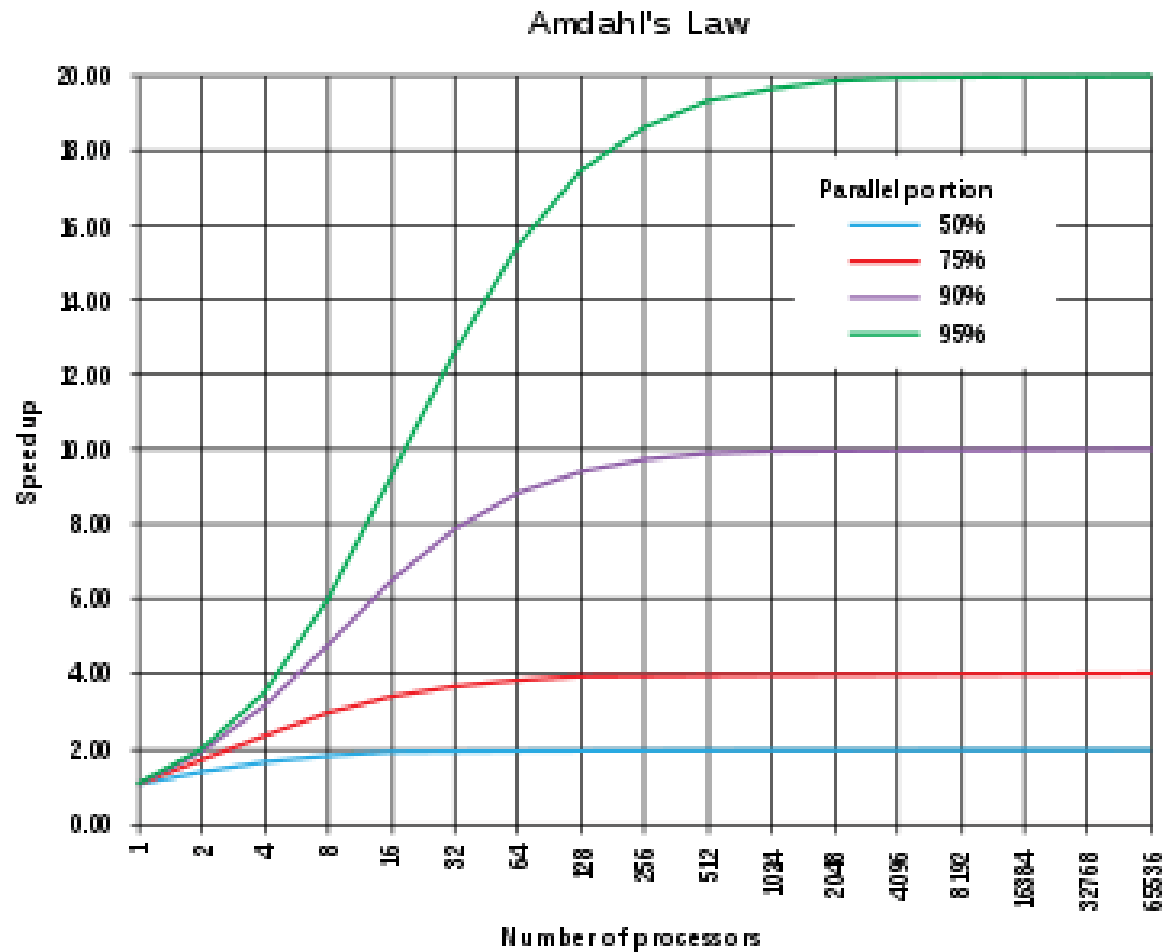
For  $P \rightarrow \text{infinity}$     $S \rightarrow 1/s$

Even if the parallel part speeds up perfectly,  
we may be limited by the sequential portion of  
code.

# Which fraction of serial code ?



# Which fraction of serial code is allowed ?



# Ahmdal law: communication overhead

- Assume that  $c(P)$  the communication time when using  $P$  processors with  $c(1)=0$

$$\rightarrow T(P) = s + p/P + c(P)$$

- Communication time may depend on many factors:
  - Network topology
  - Communication pattern
  - Message sizes
- Typical scaling of communication times:
  - Global communication, e.g. barrier:  $c(p) = k \log P$
  - Every process sending message over bus based network or serialization of communication in application code:  $c(P) = kP$

# What does it mean $k * P$ ?

$$T(1) = p + s$$



Communication model: constant fraction  $k$  for each communication among Processors

Let consider  $P=4$

$$T(4) = p/4 + s + k*4$$



$$S = \frac{1}{s + \frac{1-s}{4} + 4k}$$



# Ahmdal's law with simple communication model

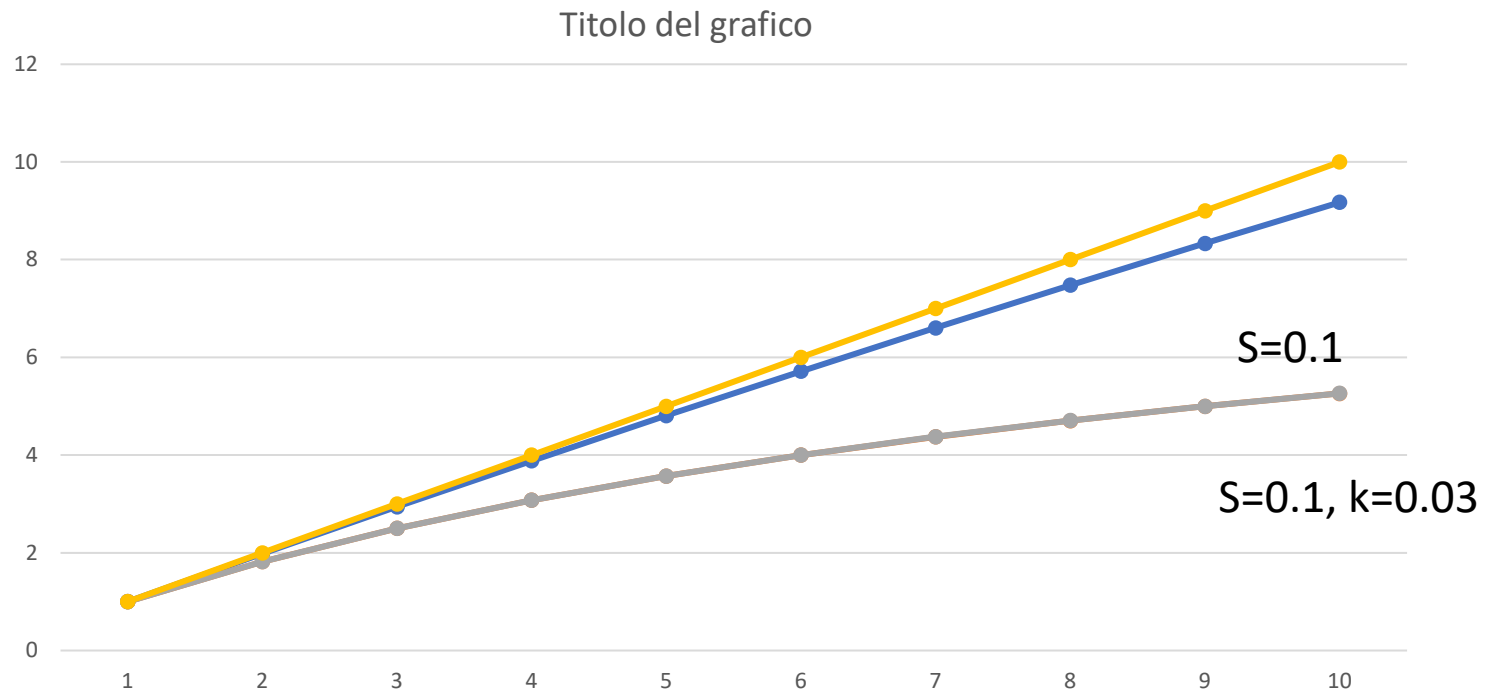
- Communication model: constant fraction  $k$  for each «communication» among processors

$$T(P) = s + p/P + kP$$

$$S(k,p) = T(1)/T(P)$$

$$S = \frac{1}{s + \frac{1-s}{P} + Pk}$$

# Which fraction of communication ?



# Large P limits

for  $P \gg 1$

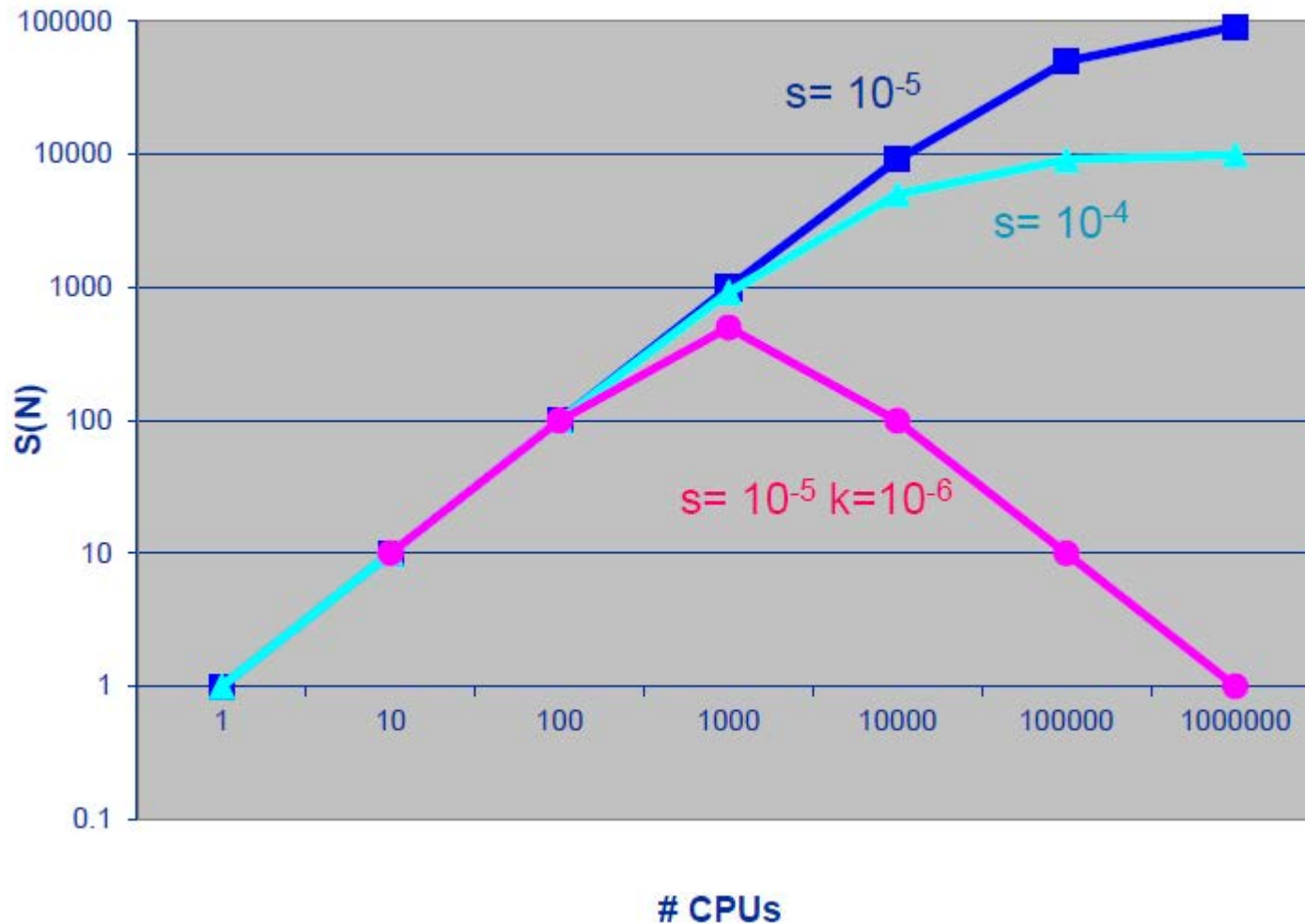
- Pure Ahmdal law :

$$S \rightarrow 1/s \text{ (Independent of } P)$$

- for  $k$  different from zero:

$$S \rightarrow 1/P^k$$

For smaller values and large P

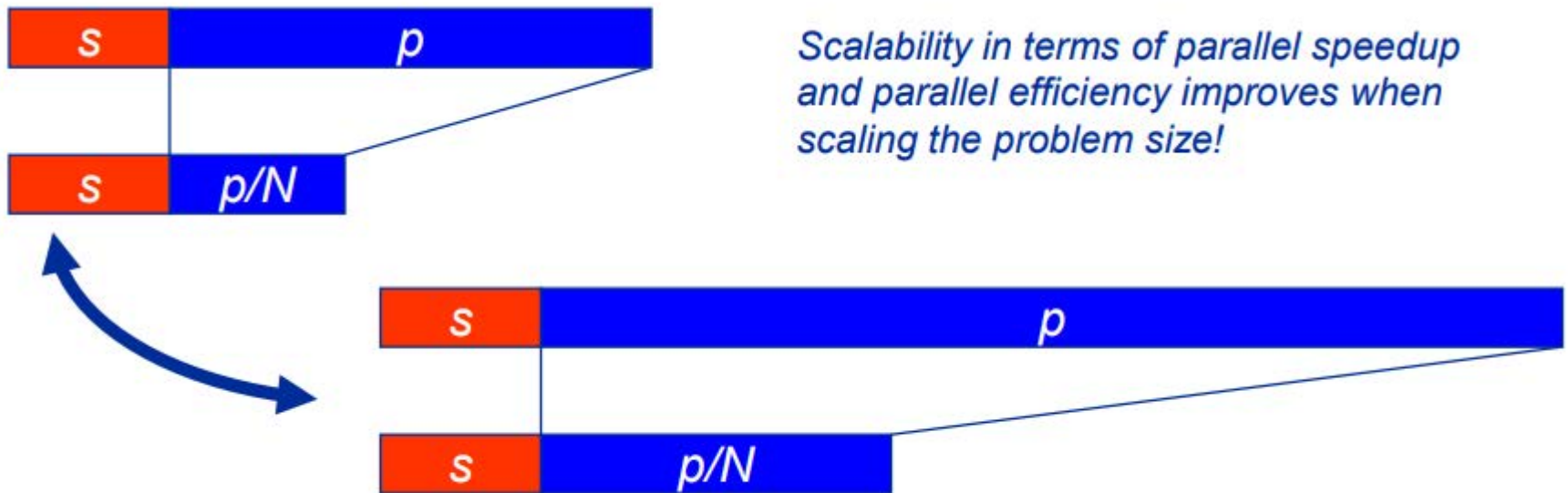


# Problem scaling

- Amdahl's Law is relevant only if serial fraction is independent of problem size, which is rarely true
- Fortunately, "The proportion of the computations that are sequential (non parallel) normally decreases as the problem size increases " (a.k.a. **Gustafon's Law**)

# The „weak scaling“ scenario

- Increasing problem size often mainly enlarges „parallel“ workload  $p$  Then Speed-up increases with



# Gustafson law

- Optimistic scenario:
  - Parallel workload increases linearly with P:

$$p \rightarrow Pp$$

$$T(P) = s + pP/P$$

$$T(P) = s + p$$

This means:

- Time remains constant when increasing parallel workload.
- Performance increase linearly with P.

# Gustafsons law

How much does it take to solve the workload of  $P$  processor on 1 processor ?

$$T_p(1) = s + Pp$$

And then:

$$S(P) = \frac{T_p(1)}{T(p)} = \frac{s+Pp}{s+p} = s + Pp = s + P(1 - s)$$

$$S(P) = P - (P-1)*s$$



# Sustained Peak performance on real scientific codes

- Blue-waters at NCSA: 22,640 AMD 6276 processors
- Theoretical peak performance: 13 Petaflops
- Sustained performance on real scientific codes...

Scientific code	Number of cores	Performance achieved(PF)	runtime (hour)
VPIC	22528	1.25	2.5
PPM	21417	1.23	~ 1
QMCPACK	22500	1.037	~1
SPECF3MD	21675	>1	Not reported
WRF	8192	0,160	<0.50

# Why performance degradation ?

- HPC system is unable to exploit all the resources all of the time
- Many different causes and many parts of the HPC are responsible all together
- At abstract level four important factors:
  - Starvation
  - Latency
  - Overhead
  - Waiting for Contention => **SLOW**

# Starvation

- Happens when sufficient work is not available at any instance in
- time to support issuing instructions to all functional units every cycle.
- Typical case:
  - Not enough parallel work for all processors/components
  - Parallel work not evenly distributed among all processors/components (load is not balanced)

# Latency

- Time it takes for information to move from one part of the system to the other.
- Typical cases:
  - Memory access
  - Data transfer between separate nodes
- Lot of tricks to hide latency (see next lectures)

# Overhead

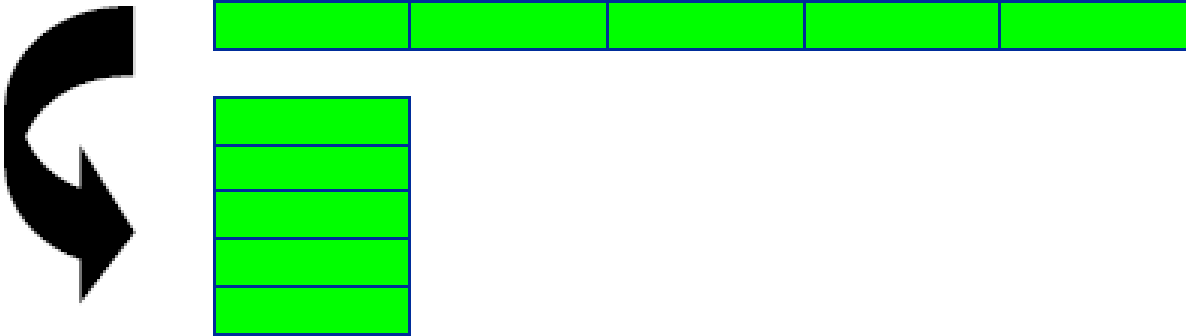
- The amount of additional work needed beyond that which is actually required to perform the computation.
- Typical cases:
  - Time to spawn and synchronize parallel tasks
  - Other kind of operation not directly associated to the computation
- The above operations steal resources to the computation and should be minimized

# Waiting for contention

- Two or more request are made at the same time on the same resource (either HW or SW)..
- Typical cases:
  - Two task writing on the same disk and/or sending message to the same memory location at the same time
  - Generally such events are not predictable and so difficult to avoid and to optimize.

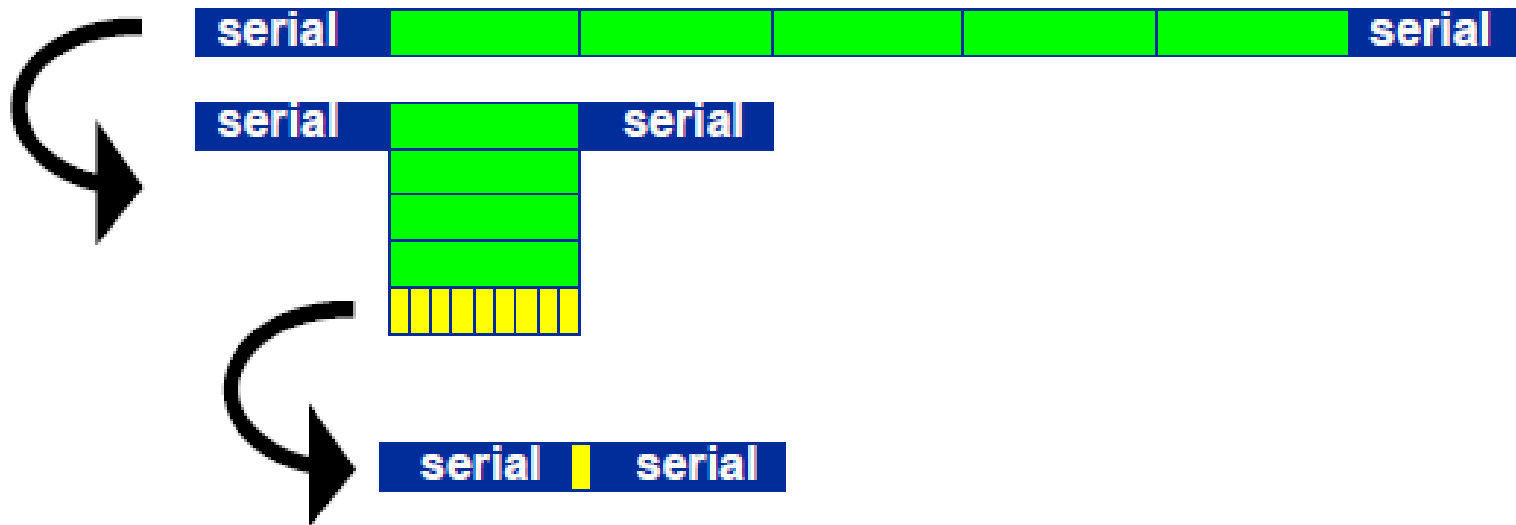
# From Ideal world ...

- All Work can be done in parallel !



# First correction..

- Serial parts limit maximum speedup





# Ugly Reality....

- Communication/synchronization /load imbalance..

