



DATA SCIENCE &  
SCIENTIFIC COMPUTING



Istituto Officina  
dei Materiali



## L03: Parallel Programming concepts

- Stefano Cozzini
- CNR-IOM and eXact lab srl

# Agenda

- HPC Concepts (part 2)
  - Parallel programming paradigms
  - Parallel programming concepts
  - Ahmdal law / Gustafson law
  - Strong/weak scalability

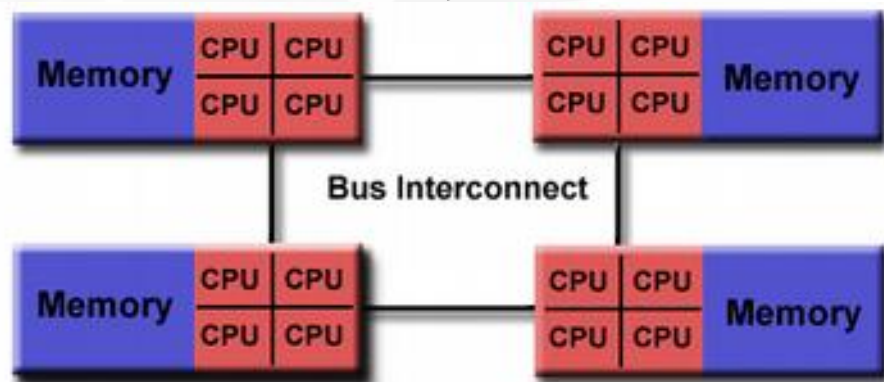
# Introduction to HPC programming principles

# Two main programming paradigms

- Dictated by the HW architecture:
  - 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

# Parallel programming

## Shared memory



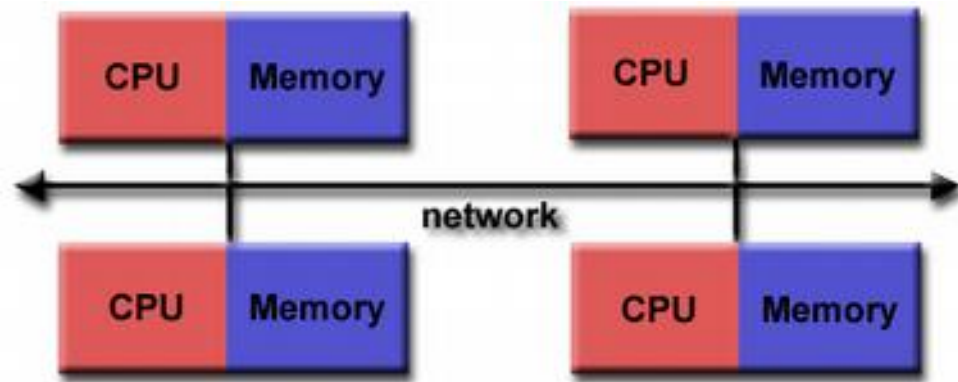
takes more time to far memor

- Pros
- 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.

## Distributed memory



- 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.

# Parallel Programming Paradigms, cont.

Programming Environments	
Message Passing	Shared memory
Standard compilers	Ad hoc compilers
Communication Libraries	Source code Directive
Ad hoc commands to run the program	Standard Unix shell to run the program
Standards: <b>MPI</b>	Standards: <b>OpenMP</b>

# How to program shared memory machine ?

- Automatic (implicit) parallelization:
  - compilers do (part of) the job for you
- Manual parallelization:
  - Insert parallel directives by yourself to help compilers
  - OpenMP THE standard
- Multi threading programming:
  - more complex but more efficient
  - use a threads library to create task by yourself
- Use already threaded libraries..

# Shared memory

- Central concept: execution thread
- A piece of the program is assigned to each thread, which runs on a core of the CPU
- Approaches
  - POSIX threads
  - OpenMP
  - Intel TBB



# Shared memory

- Simple example: loop parallelization with OpenMP

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

- To compile with gcc add '-fopenmp' to compilation line

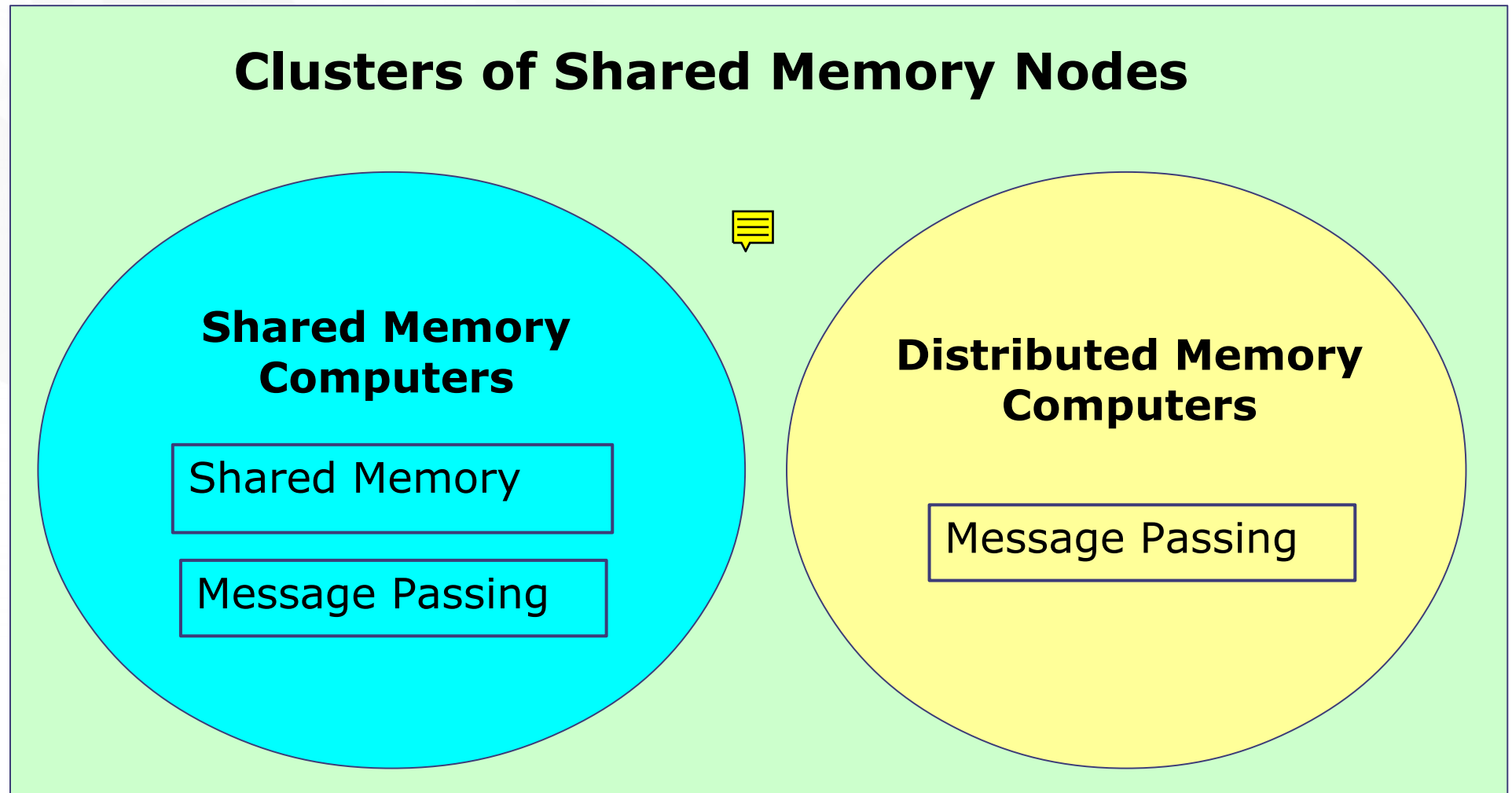
```
gcc -fopenmp mycode.c
```



# How to program using message passing ?

- 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)

# Architectures vs. Paradigms



# Parallel programming: a short summary..

Architectures	
Distributed Memory	Shared Memory
Programming Paradigms/Environment	
Message Passing	Shared Memory
Parallel Programming Models	
Domain Decomposition	Functional Decomposition

# Other paradigm are now available

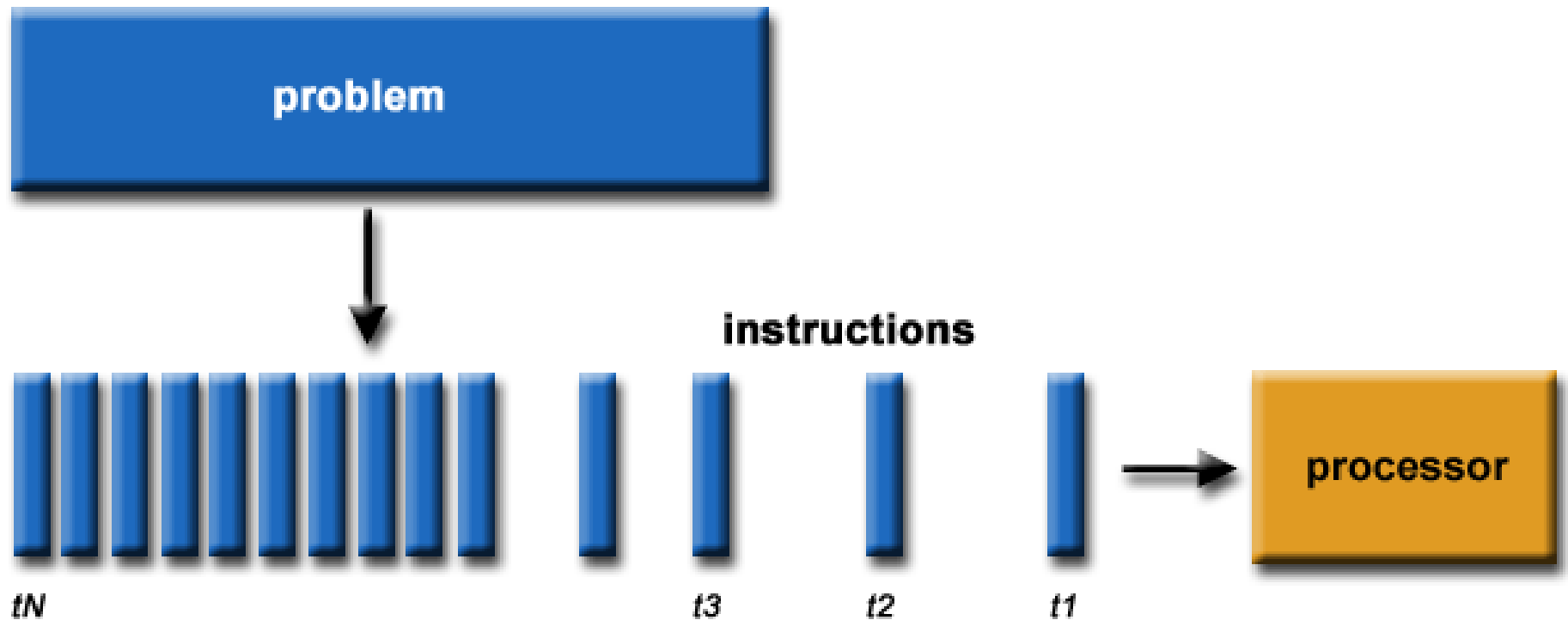
- Mixed/hybrid approach..
  - MPI + OpenMP
- Specific SDK for specific devices
  - CUDA for Nvidia GPU
- Write once run everywhere:
  - OpenCL
  - OpenACC:
    - OpenACC is about giving programmers a set of tools to port their codes to new heterogeneous system without having to rewrite the codes in proprietary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

# Principle of parallel computing

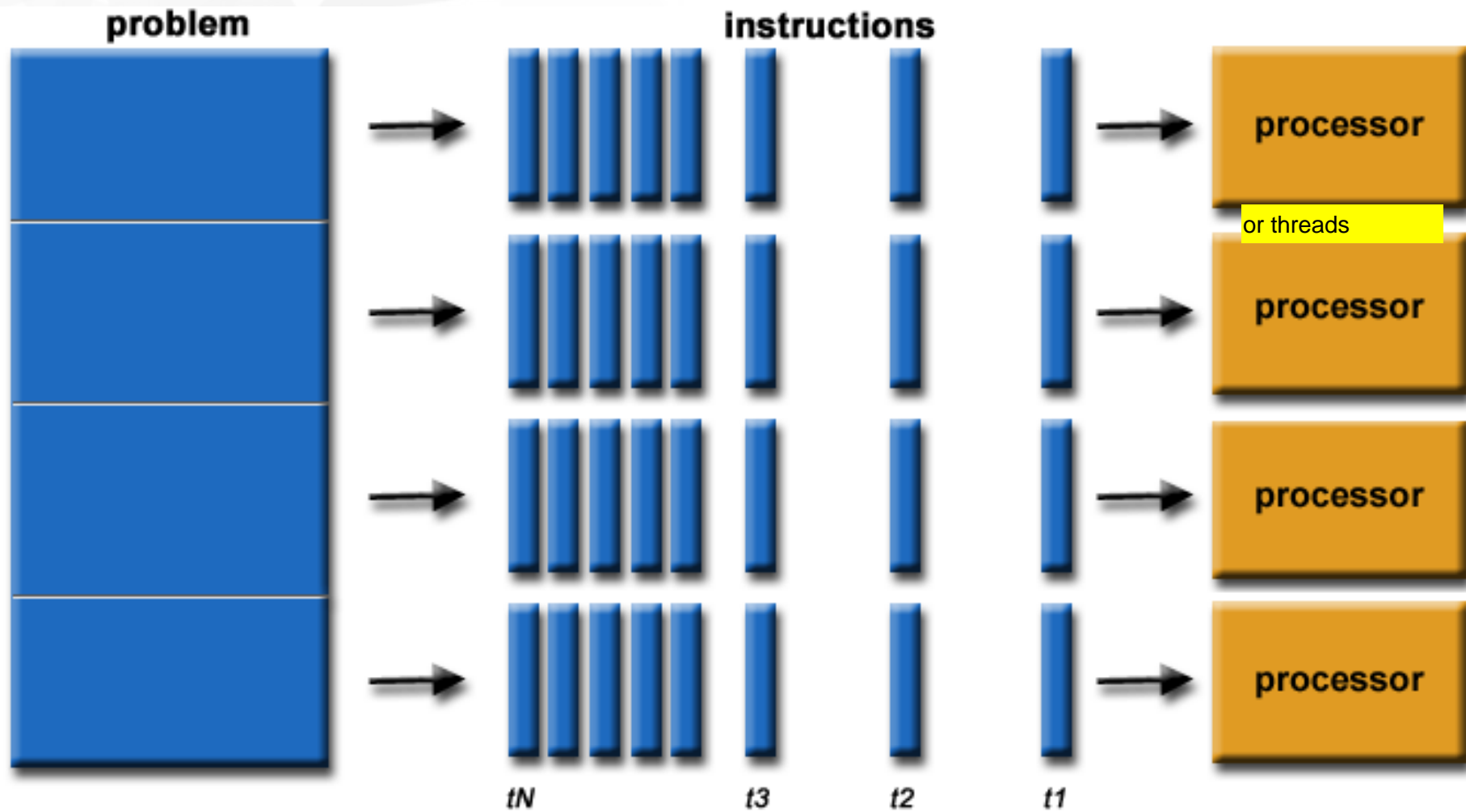
- Speedup, efficiency
  - Ahmdal Law/Gustafson Law
- Finding and exploiting parallelism
- Finding and exploiting data locality
- Load balancing
- Coordination and synchronization
- Performance modeling

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

# Serial execution

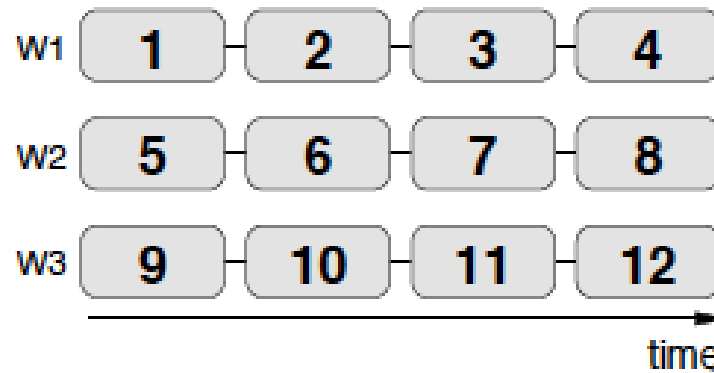
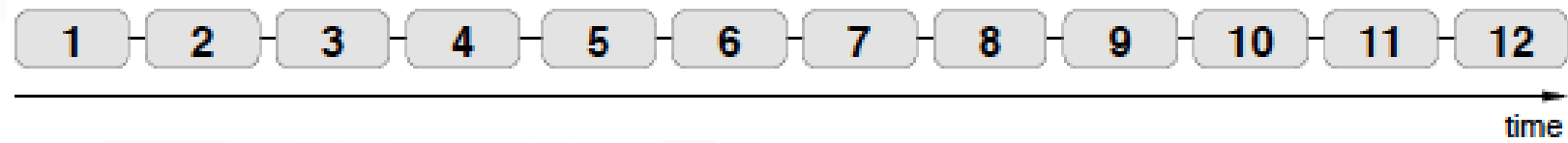


# Parallel execution



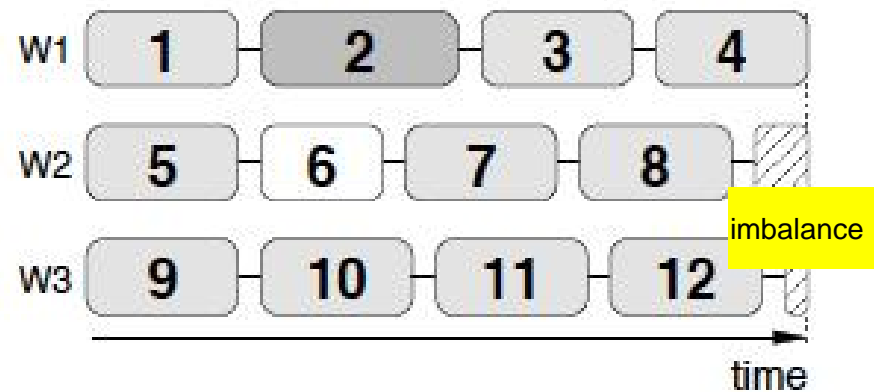


# Running in parallel



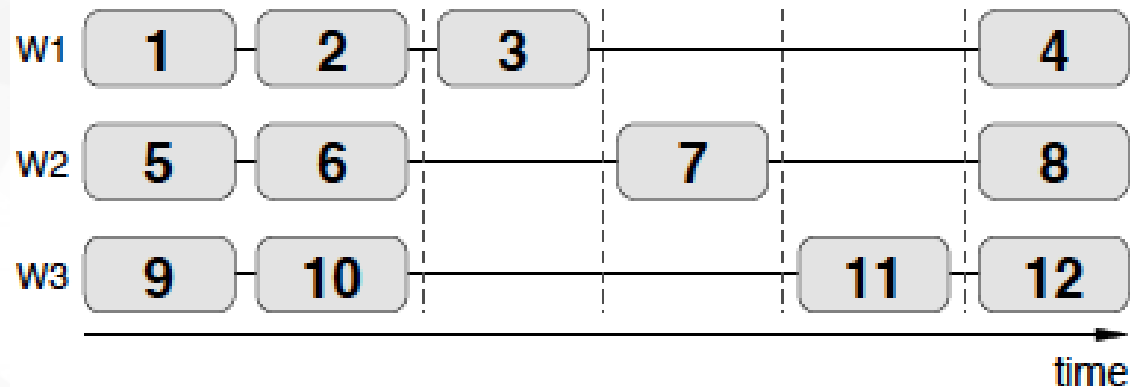
- Execution time reduces from 12 secs to 4 secs!

# Load imbalance



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

# Dependence amongst tasks



- What if tasks 3, 7 and 11 are dependent?
- Execution time increases from 4 to 6!

until 3 completed, 7 can not be run

# Introducing 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?

# Scalability



- Simple model

$$s + p = 1,$$

**s**: serial part      **p**: parallel part

- Why serial part?
  - Imperfect load balancing (some processors have more work) (Starvation)
  - Cost of communication (Latency)
  - Synchronization time (Overhead )
  - Cost of contention for resources, e.g., memory bus, I/O (Waiting )

# Scaling

- **Strong Scaling:** Keeping the problem size fixed and pushing in more workers or processors
  - Goal: Minimize time to solution for a given problem
- **Weak Scaling:** Keeping the work per worker fixed and adding more workers/processors (the overall problem size increases)
  - Goal: solve the larger problems

# Speed up

The speedup of a parallel application is

$$\text{Speedup}(p) = \text{Time}(1)/\text{Time}(p)$$

where

$\text{Time}(1)$  = execution time for a single processor

$\text{Time}(p)$  = execution time using  $p$  parallel processor

If  $\text{Speedup}(p) = p$  we have perfect speedup (also called linear scaling)

speedup compares an application with itself on one and on  $p$  processors  
more useful to compare

The execution time of the best serial application on 1 processor  
versus

The execution time of best parallel algorithm on  $p$  processors

# Scaling...

- Scaling or scalability: some sort of relation 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..



# Efficiency

- The parallel efficiency of an application is defined as
  - $\text{Efficiency}(p) = \text{Speedup}(p)/p$
- $\text{Efficiency}(p) \leq 1$
- For perfect speedup  $\text{Efficiency}(p) = 1$
- We will rarely have perfect speedup: Lack of perfect parallelism in the application or algorithm
- Understanding why an application is not scaling linearly will help will help finding ways improving the applications performance on parallel computers.

# Superlinear speedup

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

swoping -- when there is not enough pl

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

Choosing a bad “baseline” for  $T(1)$

WRONG !!!

Old serial code has not been updated with optimizations

Shrinking the problem size per processor

GOOD

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

# Amdahl's law

What is the maximum speedup for P processors?

$$\text{Speedup}(p) = T(1)/T(p)$$

$$p = 1 - s$$

$$T(p) = (1-s)*T(1)/P + s*T(1)$$

$$T(p) = T(1)*((1-s) + P*s)/P$$

assumes  
perfect  
speedup for  
parallel part

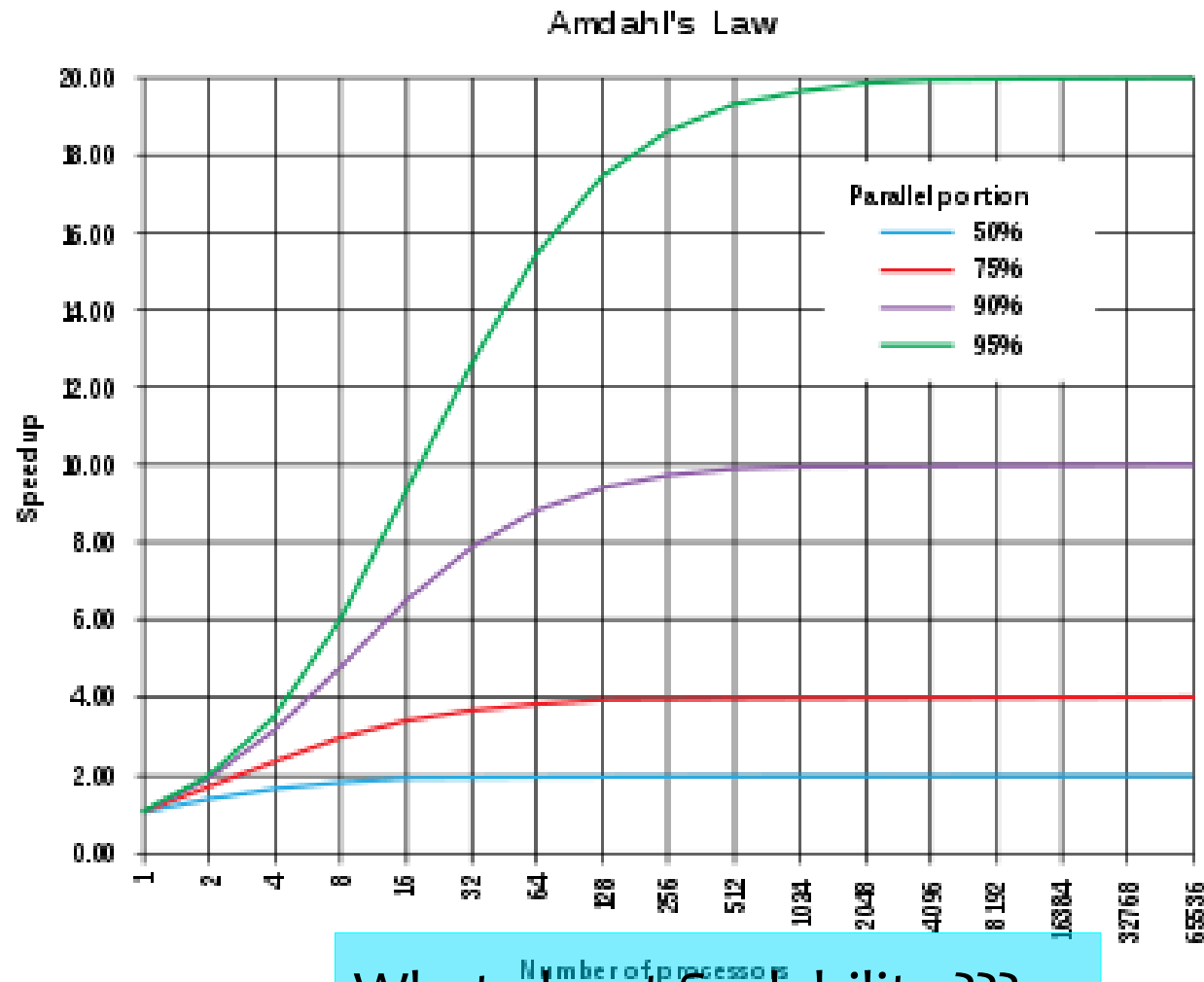
$$\begin{aligned}\text{Speedup}(p) &= P / (1 + (P-1)*s) \\ &= 1 / ((1-s)/P + s)\end{aligned}$$

$$P \rightarrow \infty \quad S = 1/s$$

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

# Amdahl's law

- Which fraction of serial code is it allowed ?



What about Scalability ???

# Ahmdal law: communication overhead

- $T_p = T_1 * s + T_1 * p/P + T_c$

- $S = T_1 / (T_1/P + T_c)$

$\Rightarrow S = P / (1 + P * (T_c/T_1))$

- P should not exceed the ratio between computing time and communication time

# Example: sum of N numbers

- Serial Algorithm :  $n-1$  operations

- $T_s = n * T_{\text{comp}}$

- $T_{\text{comp}}$  = time to compute a floating point operation

- Parallel Algorithm : master-slave

1) read  $n$  and distribute  $n$  to  **$P$  slaves**  $\implies T_{\text{read}} + (P-1) * T_{\text{comm}}$

$T_{\text{comm}}$  = time each processor takes to communicate one message,  
i.e. latency..

$T_{\text{read}}$  = time master takes to read

2)  $n/p$  sum over each processors (including master)  $\implies T_{\text{comp}}/P$

3) Slaves send partial sum  $\implies (P-1)T_{\text{comm}}$

4) Master performs final sum  $\implies T_{\text{comp}}$

$$T_p = T_{\text{comp}}(1 + n/P) + T_{\text{read}} + 2(P-1) * T_{\text{comm}}$$

# Example: sum of N numbers

- Compute Scalability:
- Estimate for your computer  $T_{\text{read}}$   $T_{\text{comp}}$   $T_{\text{comm}}$   $f$
- Formulate some scalability curve as function of **N**
- Implement a program to solve the exercise
- Check if our performance model is adequate

latency = 1.5 microsecond cycle  $T_{\text{comm}} = 1.5 * 10^{-6} T$

scalability and size of problem graph

# 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**)
- Check this on the previous example





# Gustafson law

- $T_p = T(s + p)$  whith  $s+p=1$
- $T_1 = sT + p * P * T$
- $S(P) = T_1 / T_p = (s + Pp) / (s + p) = s + Pp = p - 1 + Pp = s + P(1 - s) =$

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

# So What Is Scalability?

- to get N times more work done on N processors
- compute a fixed-size problem N times faster
  - **Strong scaling**
    - Speedup  $S = T_1 / T_N$  ; linear speedup occurs when  $S = N$
    - Can't achieve it due to Amdahl's Law (no speedup for serial parts)
- compute a problem N times bigger in the same amount of time:
  - **Weak scaling**
    - Speedup depends on the amount of serial work remaining constant or increasing slowly as the size of the problem grows
    - Assumes amount of communication among processors also remains constant or grows slowly

# Why weak scaling tends to work better..

- **Strong scaling:** fixed data/problem set; measure speedup with more processors
  - Ahmdal law
- **Weak scaling:** data/problem set increases with more processors; measure if speed(efficiency) is the same
  - Gustafson law

# References

- 
- [https://computing.llnl.gov/tutorials/parallel\\_comp/](https://computing.llnl.gov/tutorials/parallel_comp/)
- Reference 2 : chapter 2 section 2.1 2.2