deep se

dependable evolvable pervasive software engineering group

Introduction to Parallel Computing

Danilo Ardagna

Politecnico di Milano

marco.lattuada@polimi.it

danilo.ardagna@polimi.it

02/12/2024

POLITECNICO DI MILANO

Content

- Basics and Introduction What are the basic principles in PC?
- Flynn's taxonomy, shared and distributed memory architectures
- Limits and problems of parallel programming
- Examples of parallel programs

If more computational power is needed...

- How to run applications faster?
- There are 3 ways to improve performance:
  - Work Harder
  - Work Smarter
  - Get Help

- Computer Engineering recipe:
  - Use faster hardware:
    - Improve the operating speed of processors & other components
      - constrained by the speed of light, thermodynamic laws, high financial costs for processor manufacture, technical constraints
  - Optimize algorithms and techniques used to solve computational tasks
  - Use multiple computers/cores to solve a particular task
    - Connect multiple processors (or cores) together & coordinate their computational efforts

What is Parallel Computing?

- Consider your favorite computational application
  - One processor can give me results in N hours
  - Why not use N processors…
    …and get the results in just one hour?
- **Parallelism** = applying **multiple processing units (PUs)** to a **single problem**
  - Decompose the computation into many pieces
  - Assign these pieces to different processing units

  What you need to do to make it work.
- **Parallel computer (system):** a computer (system) that contains multiple processing units:
  - Each PU works on its section of the problem
  - PUs can exchange information with other PUs

Parallel vs. Serial Computers

- Two big advantages of parallel computers:
  1. total performance
  2. total memory

- Parallel computers enable us to solve problems that:
  - benefit from, or require, fast solution
  - require large amounts of memory
  - example that requires both: weather forecasting, fluid dynamic simulations, financial simulations, etc.

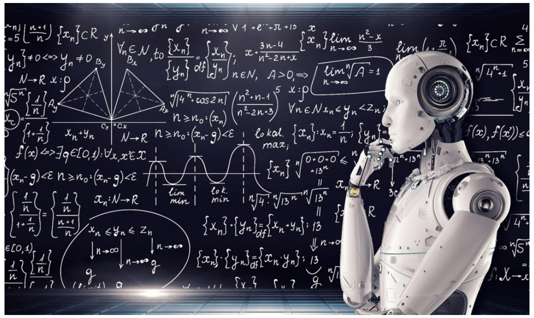
Use case examples

Parallel vs. Serial Computers

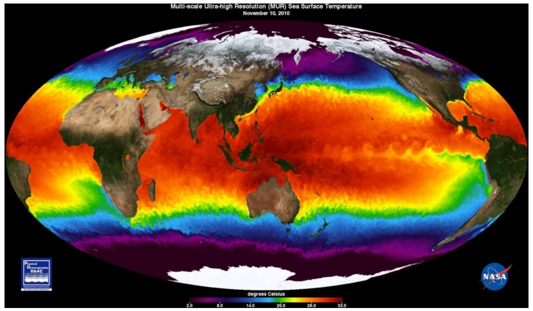
- Some benefits of parallel computing include:
  - more data points
    - bigger domains
    - better spatial resolution
    - more particles
  - more time steps
    - longer runs
    - better temporal resolution
  - faster execution
    - faster time to solution
    - more solutions in same time
    - lager simulations in real time

Examples of Parallel Applications

- Artificial Intelligence
- Weather forecast
- Vehicle design and dynamics
- Analysis of protein structures
- Human genome work
- Astrophysics
- Earthquake wave propagation
- Molecular dynamics
- Climate, ocean modeling
- Imaging and Rendering
- Petroleum exploration
- Database query
- Ozone layer monitoring
- Natural language understanding
- Study of chemical phenomena
- And many other scientific and industrial simulations



shutterstock.com • 642071956



Flynn's Taxonomy

- Defines the types of parallel architectures
- Based on the number of instruction streams and data streams
- A stream simply means a sequence of items (data or instructions)

Flynn's Taxonomy

	Data stream	
Instruction stream	SISD	SIMD
	(MISD)	MIMD

- SISD: Single instruction, single data
  - Sequential processing
- SIMD: Single instruction, multiple data
- MISD: Multiple instructions, single data
  - Nonexistent, just listed for completeness

← doesn't exist

- MIMD: Multiple instructions, multiple data
  - Most common and general parallel machine

Flynn’s Taxonomy

Data stream

Instruction stream

SISD	SIMD
(MISD)	MIMD

- SISD: Single instruction, single data

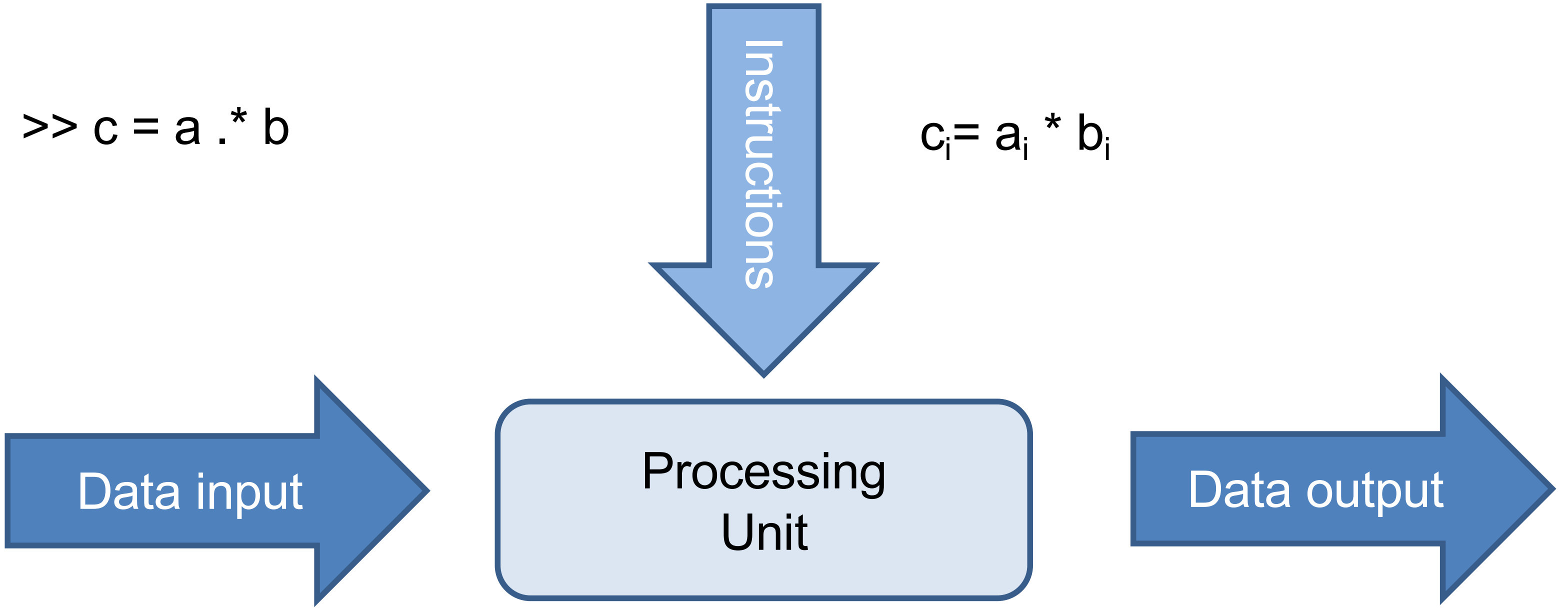
In what will follow we introduce simplifications and abstraction: Flynn’s instruction streams means the stream of assembler instructions!!!

  - Most common and general parallel machine

SISD: Conventional computer

`>> c = a .* b`

$c_i = a_i * b_i$



a_0, a_1, a_2, a_3
b_0, b_1, b_2, b_3

$c_0 = a_0 * b_0$
$c_1 = a_1 * b_1$
$c_2 = a_2 * b_2$
$c_3 = a_3 * b_3$

c_0, c_1, c_2, c_3

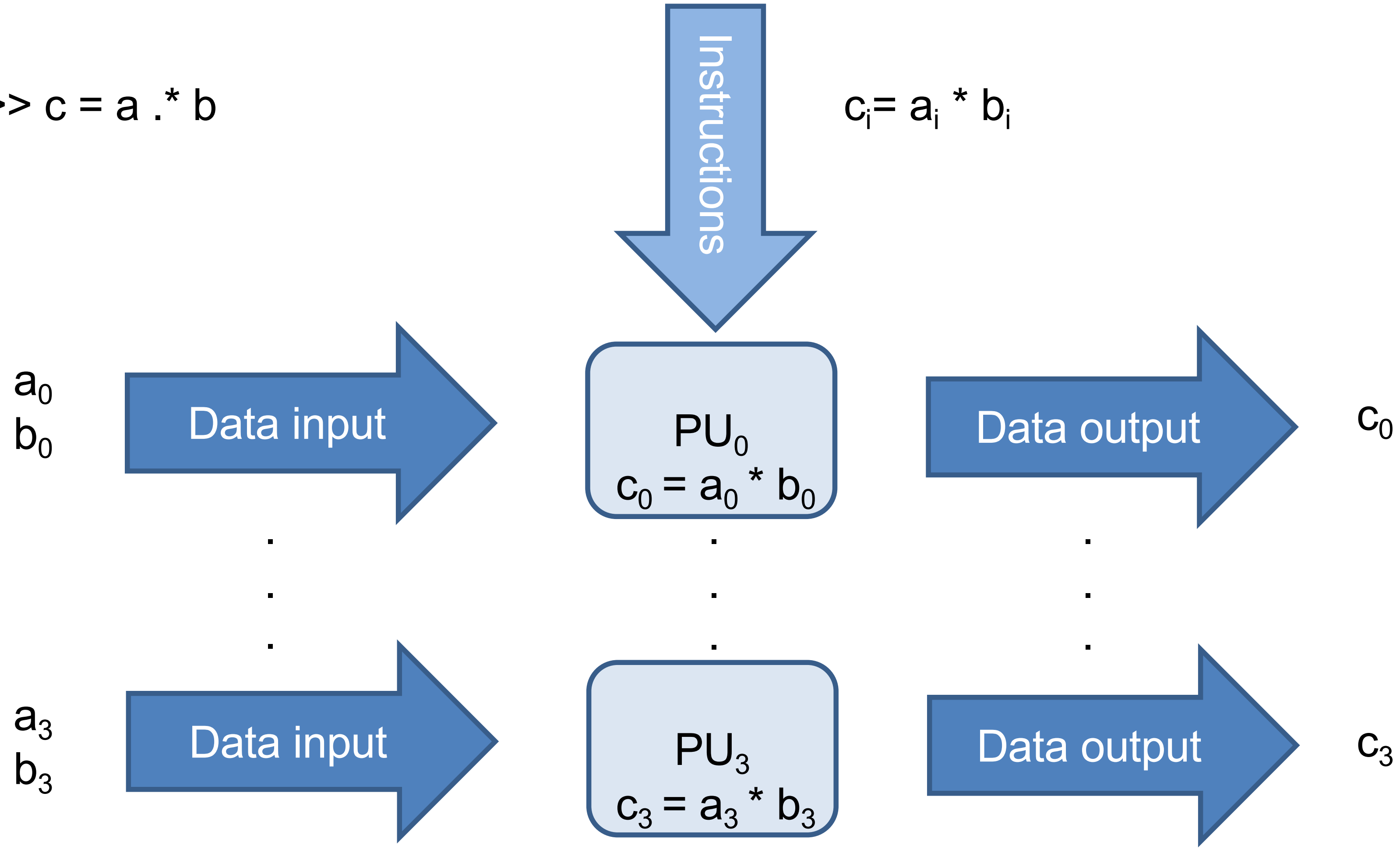
- Speed is limited by the rate at which computer can transfer information internally

SIMD

"single instruction multiple data"

>> c = a .* b

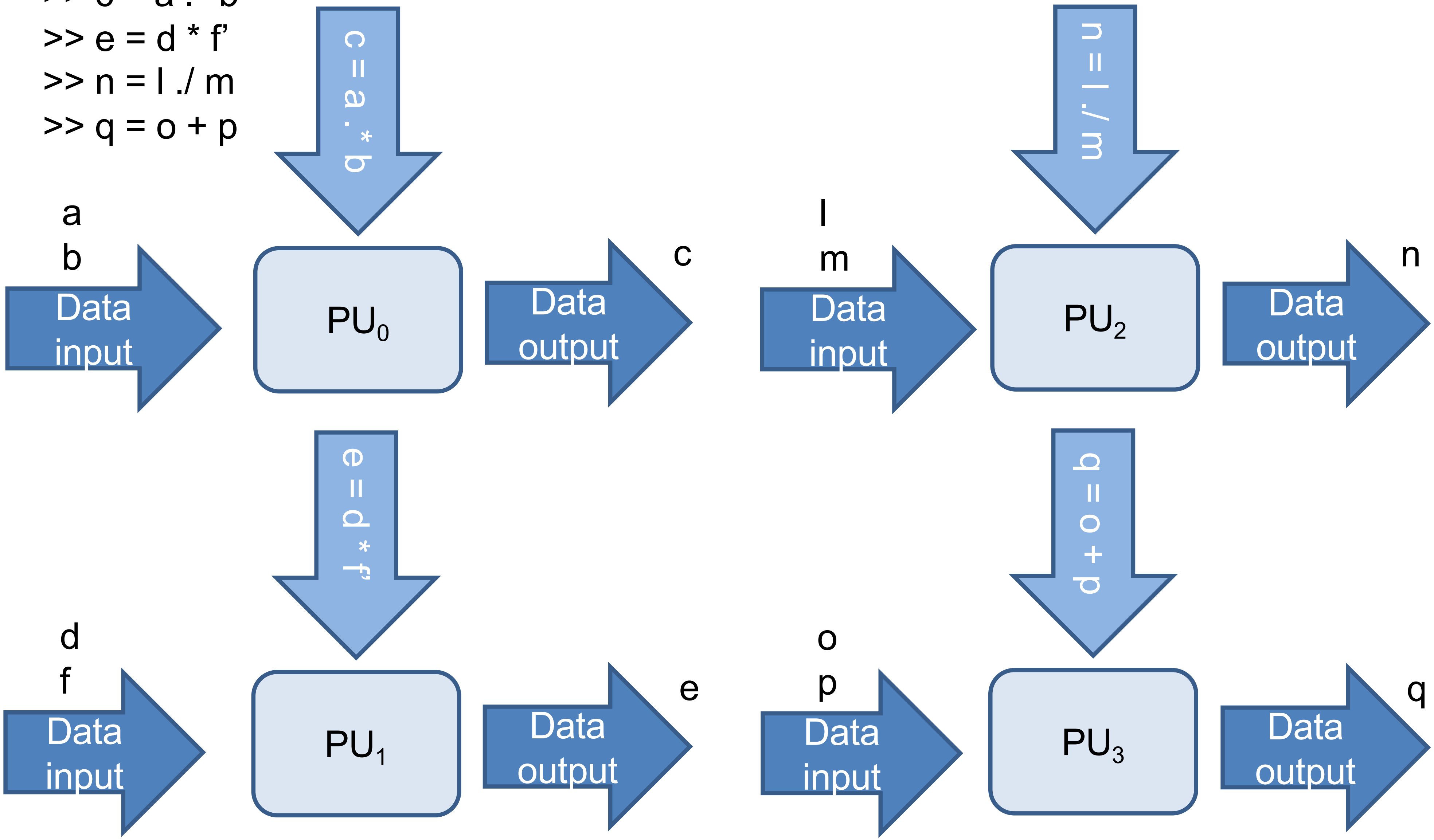
$c_i = a_i * b_i$



- Only one instruction is executed on different data simultaneously
- SIMD relies on the regular structure of computations

MIMD "Multiple Instruction Multiple data"

```
>> c = a .* b
>> e = d * f'
>> n = l ./ m
>> q = o + p
```



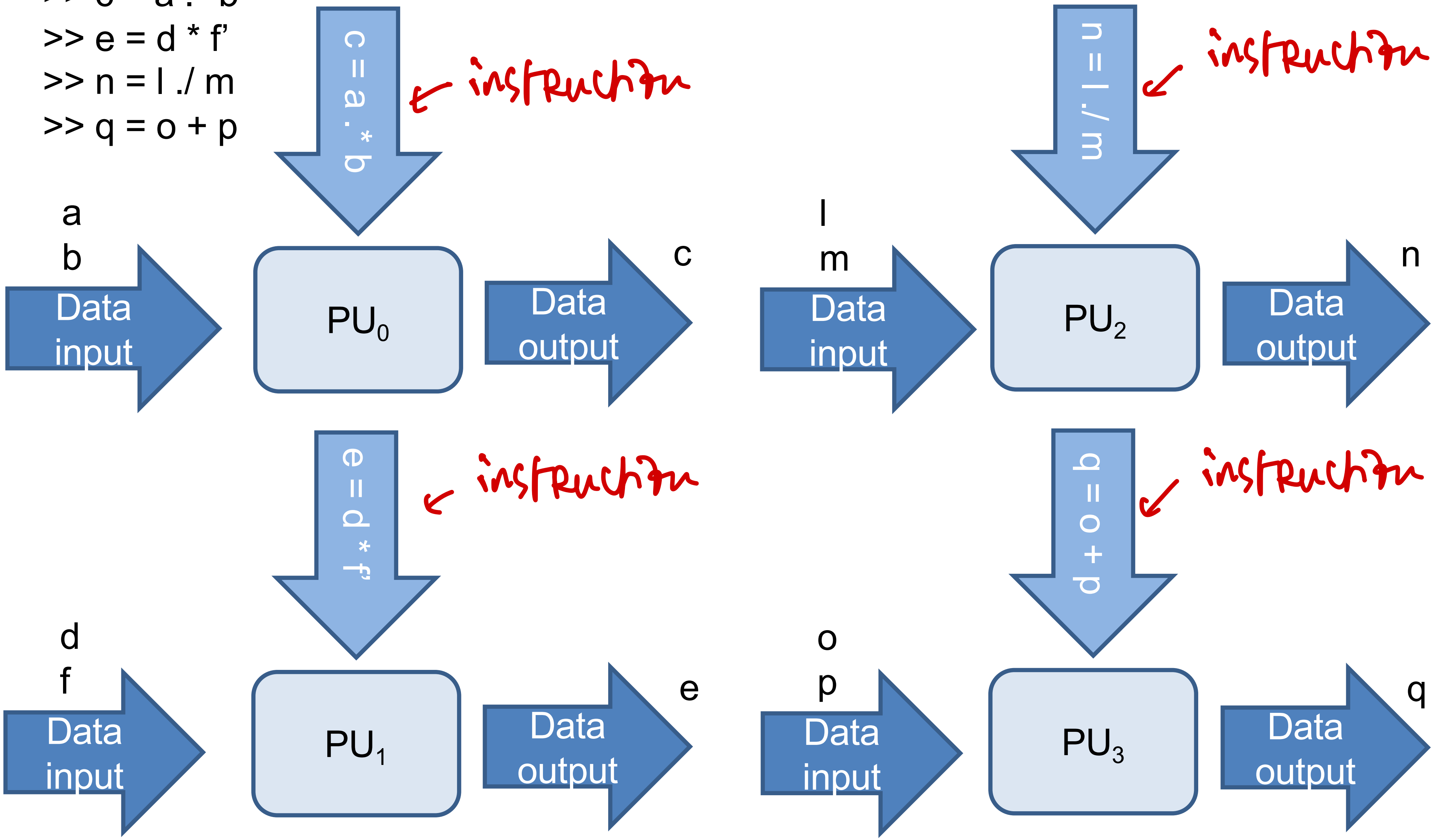
MIMD

- In contrast to SIMD processors, MIMD processors can execute different programs on different processors
- A variant of this, called single program multiple data streams (SPMD) executes the same program on different processors
- SPMD and MIMD are closely related in terms of programming flexibility and underlying architectural support
- MIMD most widely used architectural model today
- Issues:
  - Data distribution and dependency
  - Synchronization
  - Communication cost

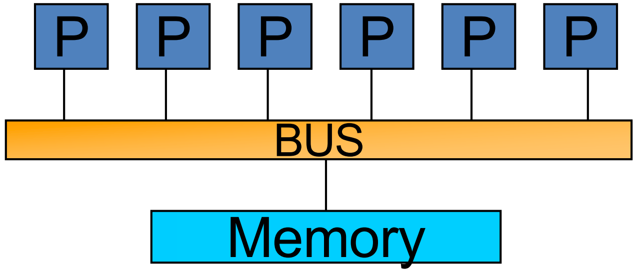
MIMD

- Issues:
  - Data distribution and dependency
  - Synchronization
  - Communication cost

```
>> c = a .* b
>> e = d * f'
>> n = l ./ m
>> q = o + p
```



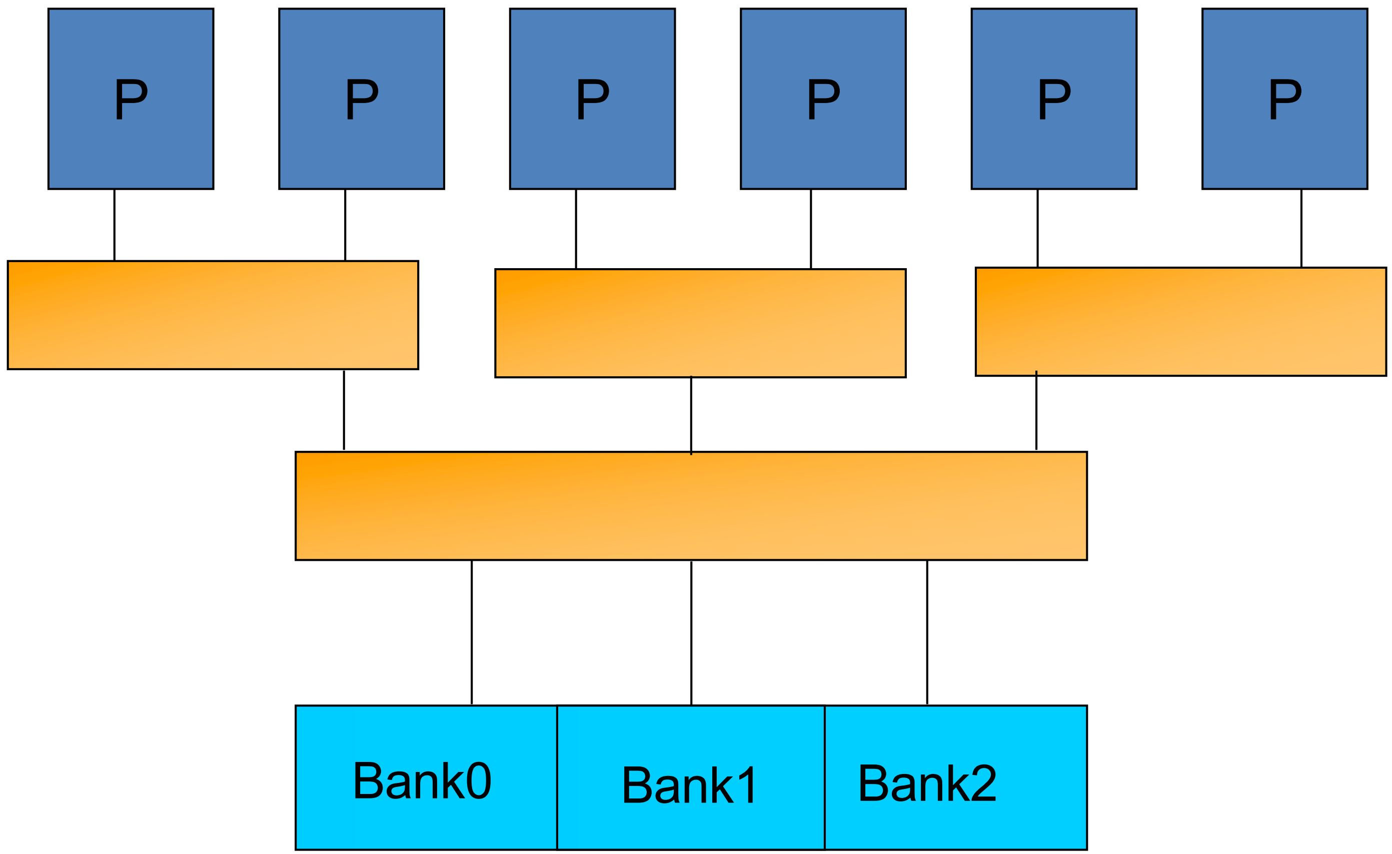
MIMD today: Shared and Distributed memory



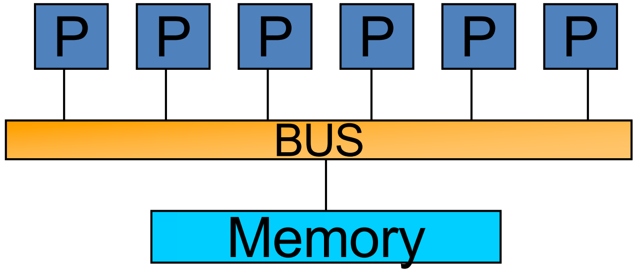
Shared memory

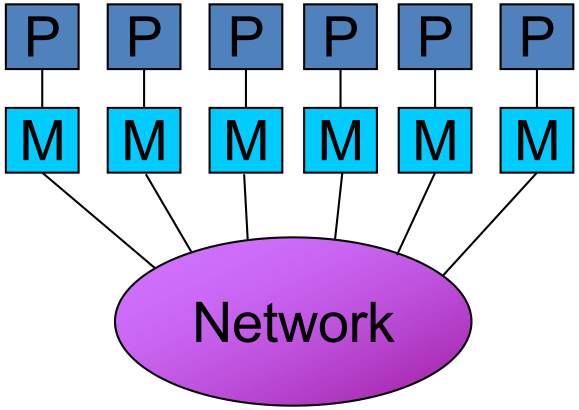
- Single address space. All processors have access to a pool of shared memory
- Processor-to-processor data transfers are done using shared areas in memory
- Scalability limits
- Methods of memory access:
  - Bus
  - Crossbar

Shared memory with crossbar



MIMD today: Shared and Distributed memory





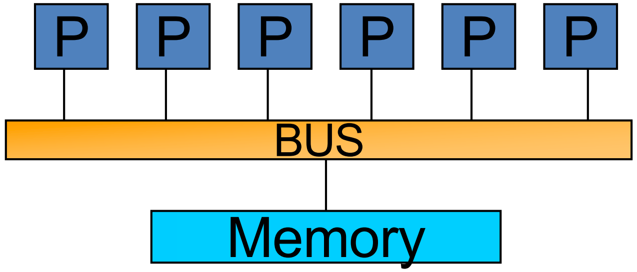
Shared memory

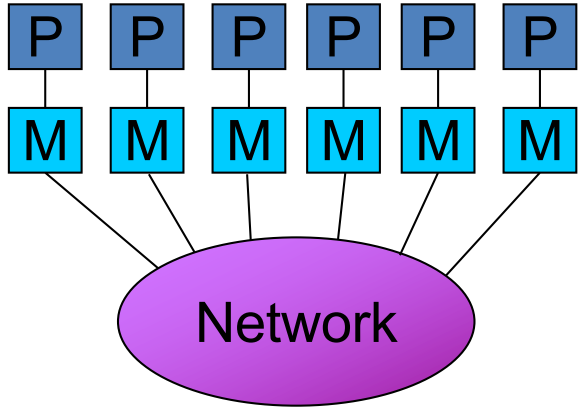
- Single address space. All processors have access to a pool of shared memory
- Processor-to-processor data transfers are done using shared areas in memory
- Scalability limits
- Methods of memory access:
  - Bus
  - Crossbar

Distributed memory

- Each processor has its own local memory
- Must do message passing to exchange data between processors
- High scalability, but load balancing issues exist and I/O is difficult

MIMD today: Shared and Distributed memory





Shared memory

- Single address space. All processors have access to a

Distributed memory

- Each processor has its own local memory

- Focus of this course: MPI - Message passing interface
- MPI executed both in shared and distributed memory
- Model MIMD but most frequently you will consider SPMD exploiting data parallelism

Parallel Computing – Real Life Scenario

- Parallel processing allows to accomplish a task faster by dividing the work into a set of subtasks assigned to multiple workers
- Assigning a set of books to workers is **task partitioning**. Passing of books to each other is an example of **communication** between subtasks

- Some problems may be completely serial; e.g., digging a post hole. Poorly suited to parallel processing
- All problems are not equally amenable to parallel processing

Limits and problems of Parallel Computing

- Not all the algorithms can be parallelized
  - Not all the problems can be solved in a parallel way
- Theoretical Upper Limits (see next slides)
  - Amdahl's Law
- Practical Limits
  - Load balancing
  - Non-computational sections (I/O, system ops, etc.)
- Different approach than sequential programming
  - Rethink the algorithms
  - Re-write code

Theoretical Upper Limits to Performance

- All parallel programs contain:
  - Serial sections
  - Parallel sections
- Serial sections, when work is duplicated or no useful work done (waiting for others), limit the parallel effectiveness
  - Lot of serial computation gives bad speedup
  - No serial work “allows” perfect speedup
- **Speedup** is the ratio of the time required to run a code on one processor to the time required to run the same code on multiple (N) processors: Amdahl’s Law states this formally

see next slides

Amdahl's Law

- Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors.
  - Effect of multiple processors on run time

$$t_n = \left(f_p / N + f_s\right) t_1$$

  - Effect of multiple processors on speed up ($S = t_1/t_n$)

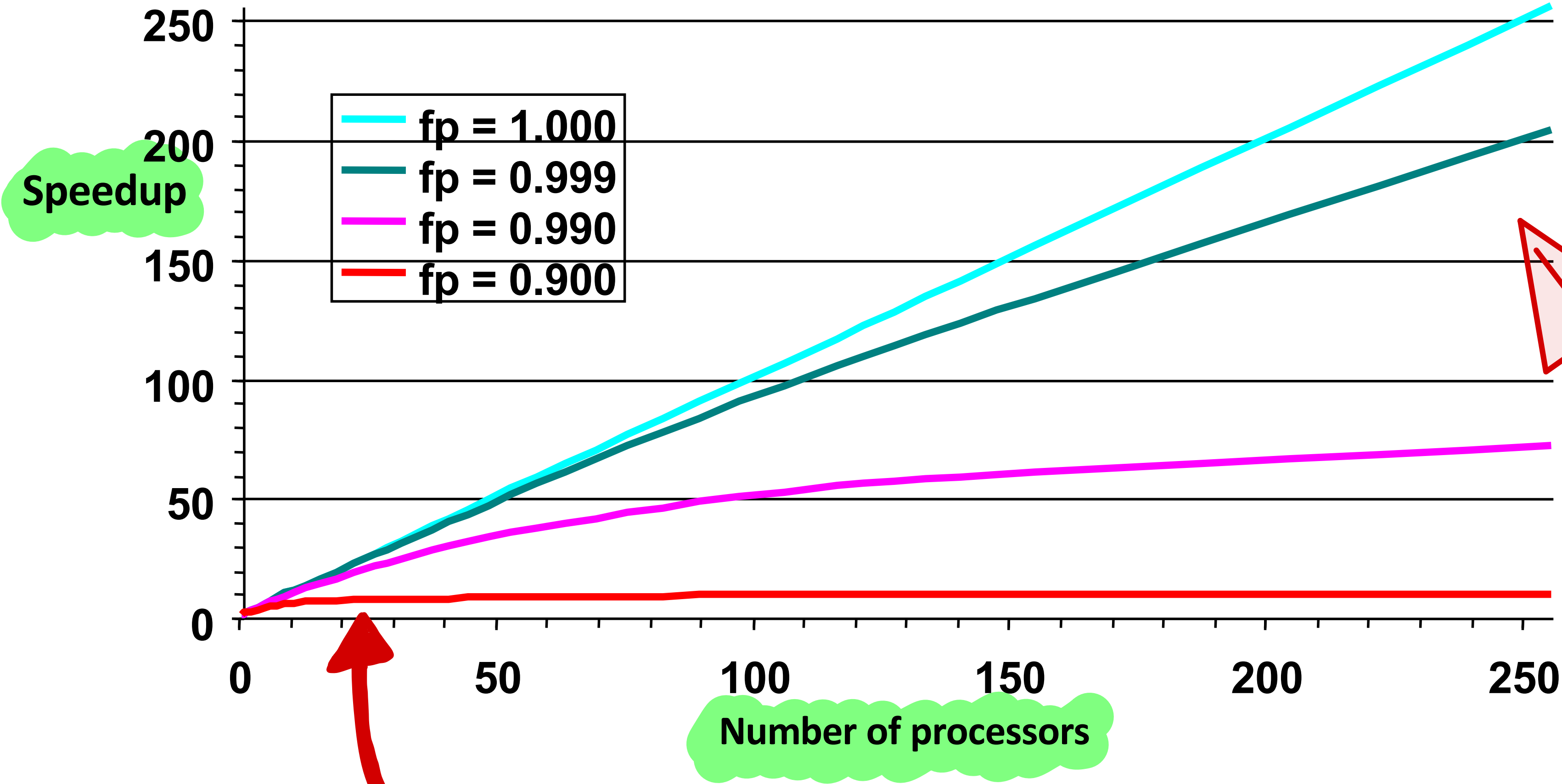
$$S = \frac{1}{f_s + f_p / N}$$

speedup $= t_1 / t_n$.

  - Where
    - f_s = serial fraction of code
    - f_p = parallel fraction of code
    - N = number of processors
    - t_n = time to run on N processors

Illustration of Amdahl's Law

It takes only a small fraction of serial content in a code to degrade the parallel performance



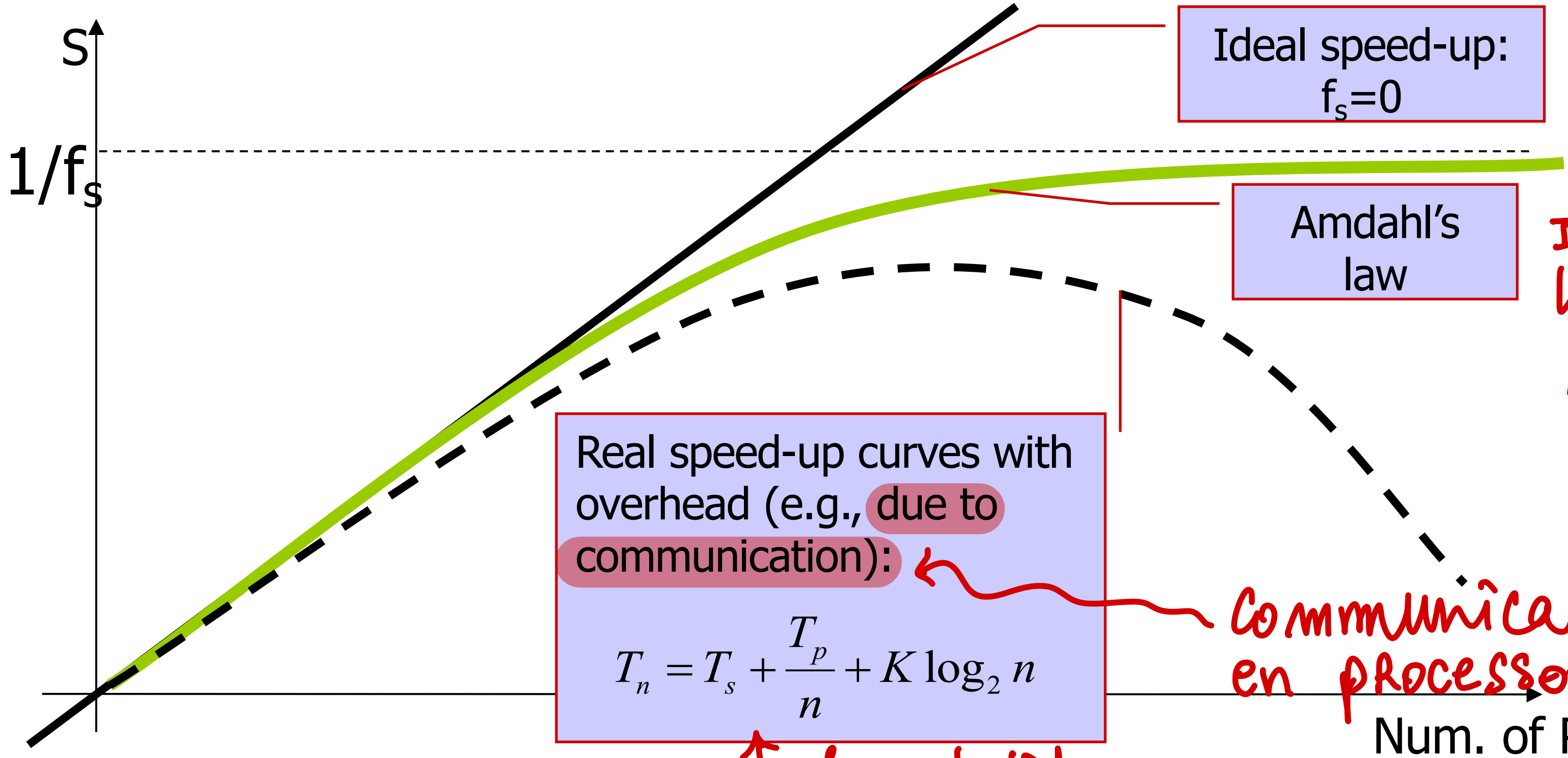
même avec seulement 10% du code en série.

Performance dégradée !

avec 90% du code en // : le série est prépondérant.

Amdahl's Law vs. Reality

Amdahl's Law provides a theoretical upper limit on parallel speedup assuming that there are no parallelization overhead. In reality, overhead will result in a further degradation of performance



It is a practical limitation of Amdahl's law.

Communication between processors in parallel computing.

cf next slide.

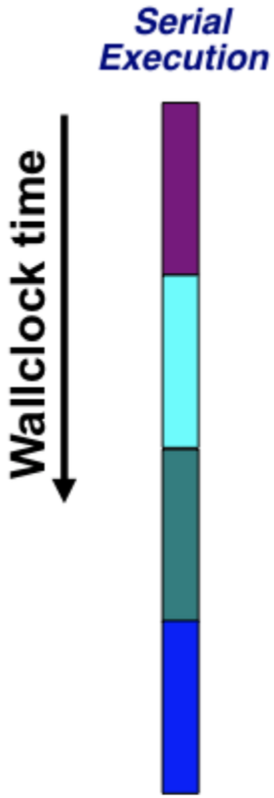
Sources of Parallel Overhead

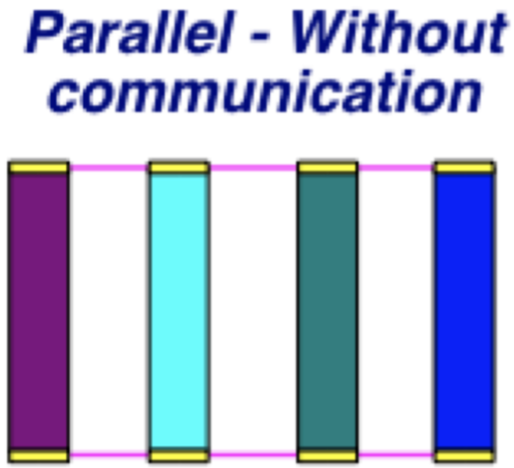
- **Interprocessor communication**: Time to transfer data between processors is usually the most significant source of parallel processing overhead

- **Load imbalance**: In some parallel applications it is impossible to equally distribute the subtask workload to each processor. So at some point all but one processor might be done and waiting for one processor to complete

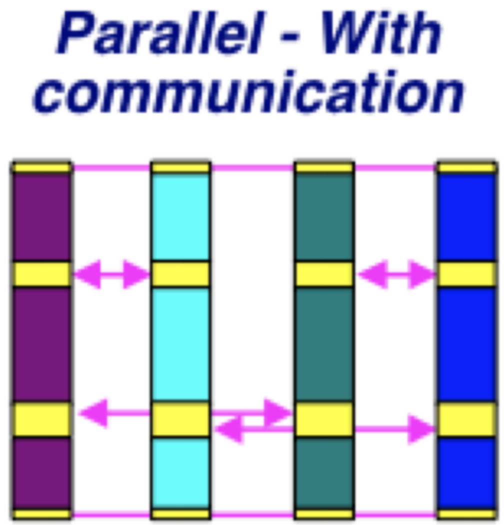
- **Extra computation**: Sometimes the best sequential algorithm is not easily parallelizable and one is forced to use a parallel algorithm based on a poorer but easily parallelizable sequential algorithm. Sometimes repetitive work is done on each of the N processors which leads to extra computation

Communication effect





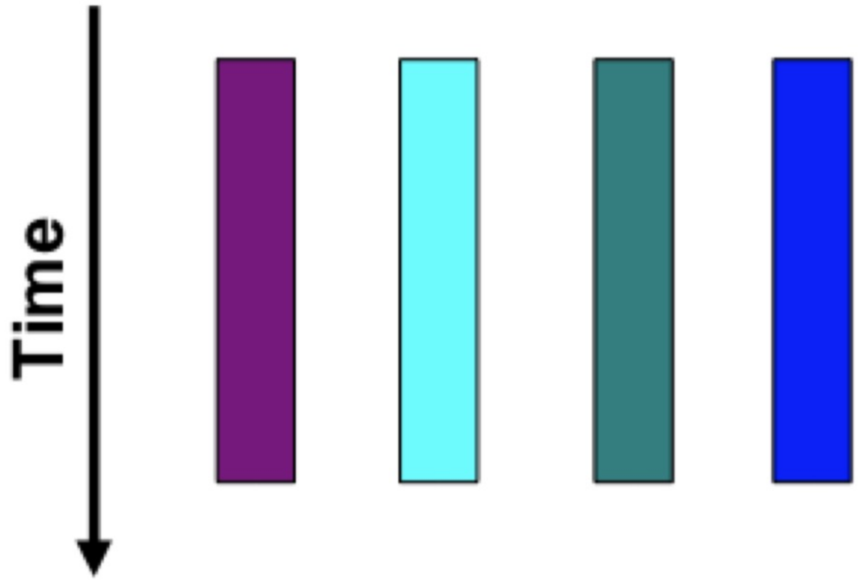
- Embarrassingly parallel: 4x faster
- Wallclock time is ¼ of serial wallclock time



- Additional communication
- Less than 4x faster
- Consumes additional resources
- Wallclock time is more than ¼ serial wallclock time

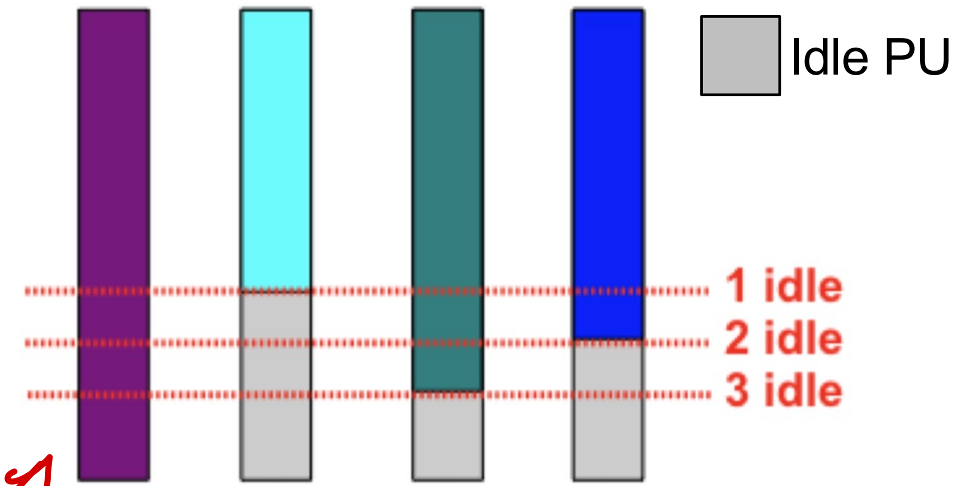
Load imbalance effect

- Perfect balance



- All PUs finish in the same amount of time
- No PU is idle

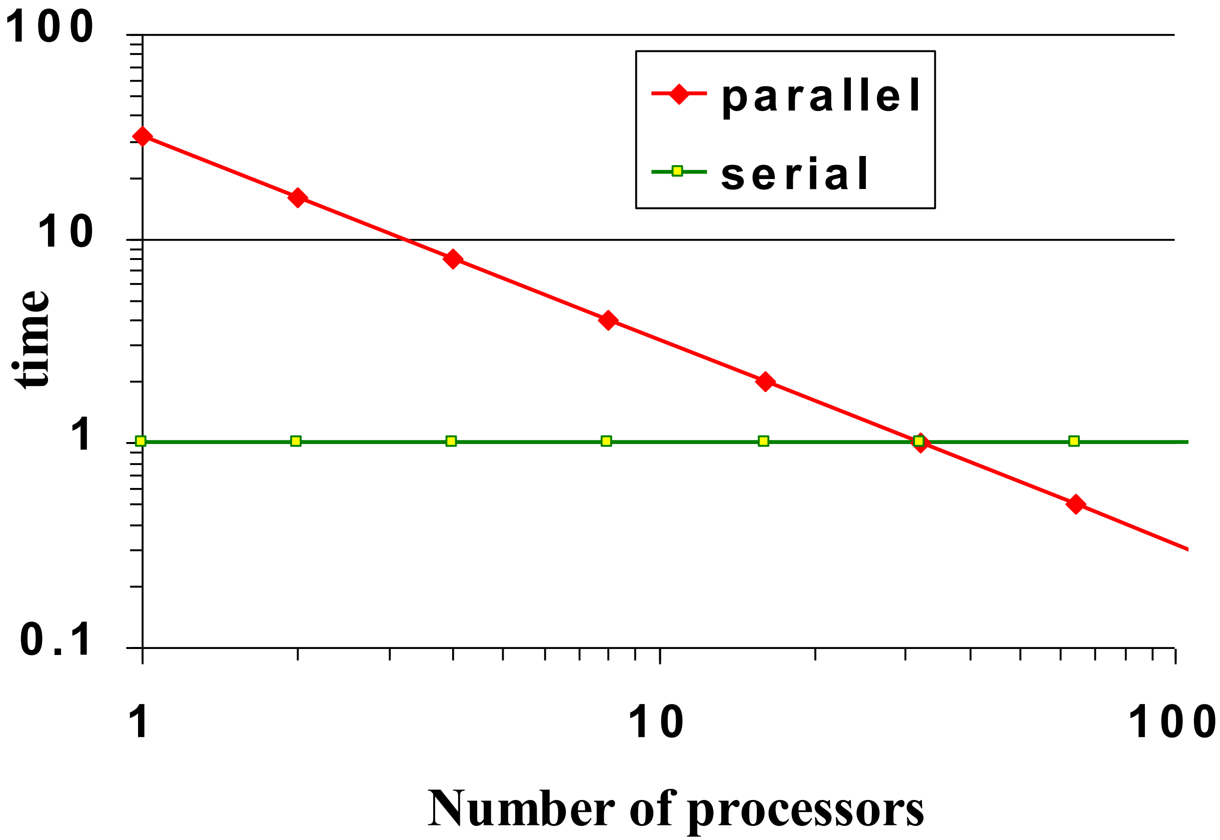
- Load imbalance



- Different PUs need a different amount of time to finish their task
- Total wall clock time increases
- Program does not scale well

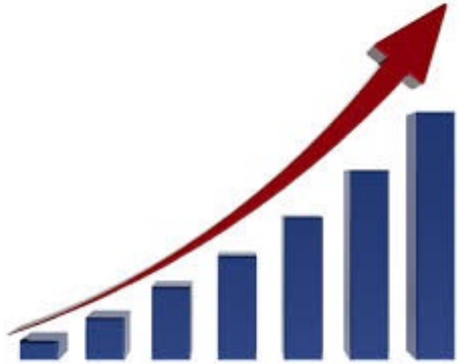
Here, only one IU works, others wait!

Serial Performance



In this case, the parallel code achieves perfect scaling, but does not match the performance of the serial code until 32 processors are used

Superlinear speedup



- In practice a speedup greater than N (on N processors) is called super-linear speedup

- This is observed due to
  - Non-optimal sequential algorithm
  - Sequential problem may not fit in one processor's main memory and require slow secondary storage, whereas on multiple processors problem fits in main memory of N processors

The 4 Horsemen of the Apocalypse (SLOW)

- Starvation
  - Not enough work to do due to insufficient parallelism or poor load balancing among distributed resources
- Latency
  - Waiting for access to memory or other parts of the system
- Overhead
  - Extra work that must be done to manage program concurrency and parallel resources, rather than the real work you want to perform
- Waiting for Contention
  - Delays due to fighting to use a shared resource. Network bandwidth is a major constraint

Performance comes at a price: complexity

- Is it worth your time to rewrite your application?
  - Do the CPU requirements justify parallelization?
  - Will the code be used just once?

- Writing effective parallel applications is difficult

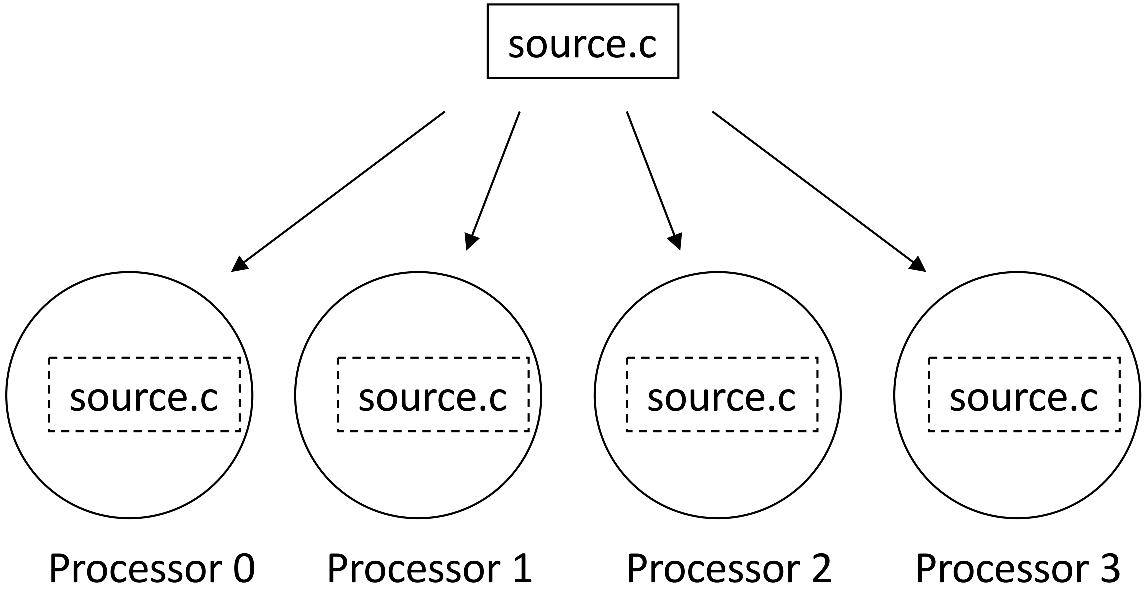
- Performance characteristics of applications change and become architecture dependent

- Debugging becomes more of a challenge

Examples of Parallel Programs

Single Program, Multiple Data (SPMD)

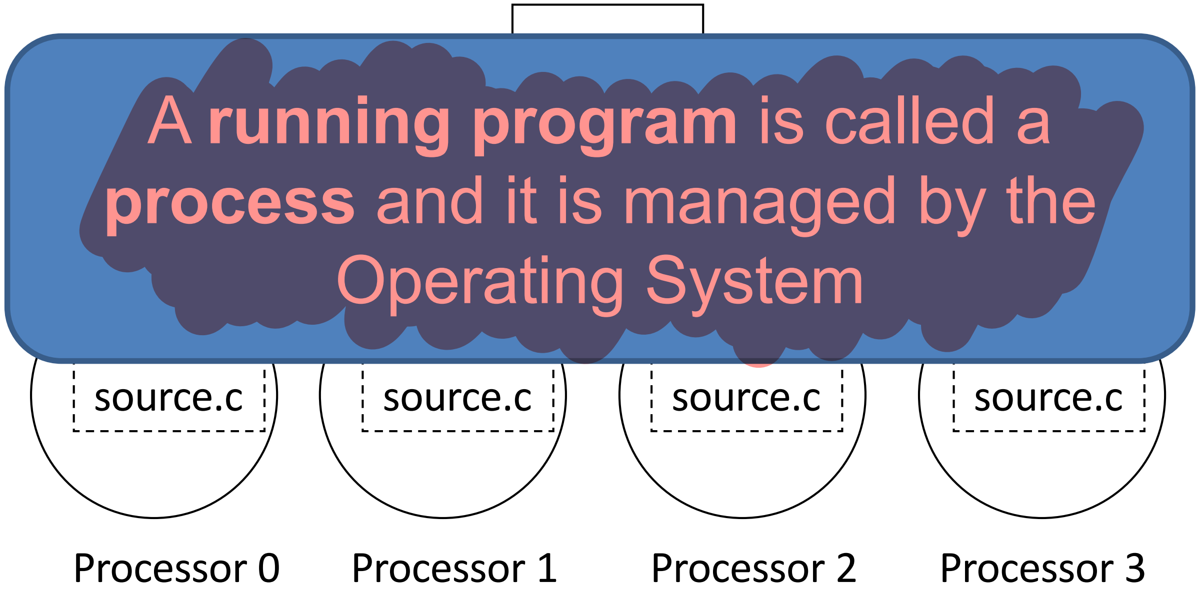
- SPMD: dominant programming *model*
  - Only a single source code is written
  - Code can have conditional execution based on which processor is executing the copy
  - All copies of code are started simultaneously and communicate and synch with each other periodically



Single Program, Multiple Data (SPMD)

- SPMD: dominant programming *model*
  - Only a single source code is written
  - Code can have conditional execution based on which processor is executing the copy
  - All copies of code are started simultaneously and communicate and synch with each other periodically

A **running program** is called a **process** and it is managed by the Operating System



Basics of Data Parallel Programming

One code will run on 2 CPUs

Program has array of data to be operated on by 2 CPU so array is split into two parts.

```
program:
…
if CPU=0 then
   low_limit=1
   upper_limit=50
elseif CPU=1 then
   low_limit=51
   upper_limit=100
end if
do I = low_limit,
upper_limit
   work on A(I)
end do
...
end program
```

CPU 0

```
program:
…
if CPU=0 then
   low_limit=1
   upper_limit=50
elseif CPU=1 then
   low_limit=51
   upper_limit=100
end if
do I = low_limit,
upper_limit
   work on A(I)
end do
...
end program
```

CPU 1

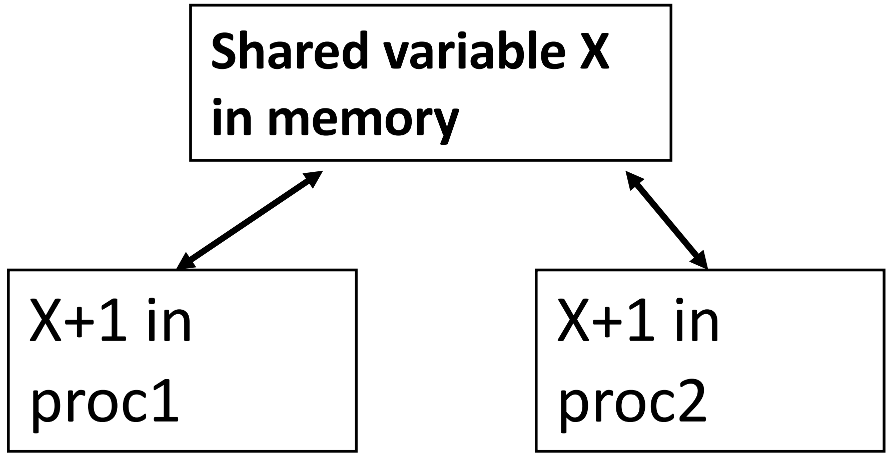
```
program:
…
if CPU=0 then
   low_limit=1
   upper_limit=50
elseif CPU=1 then
   low_limit=51
   upper_limit=100
end if
do I = low_limit,
upper_limit
   work on A(I)
end do
...
end program
```

Accessing Shared Variables

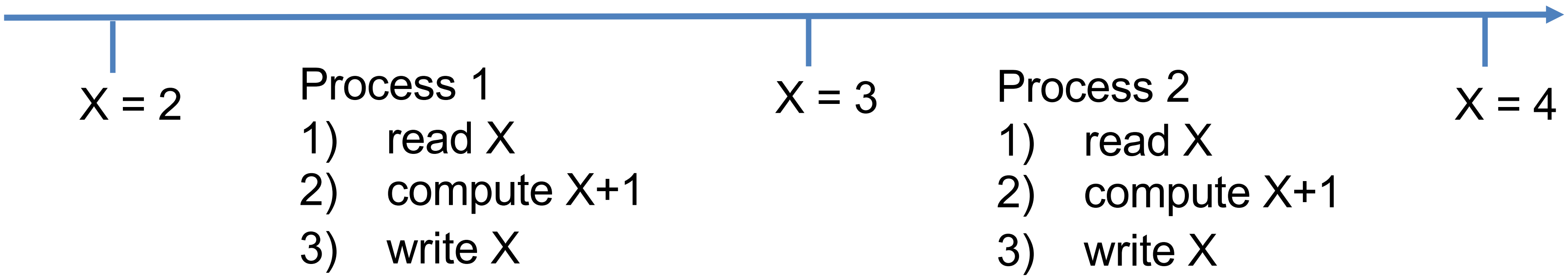
- If multiple processors want to write to a shared variable at the same time there may be conflicts :

Process 1 and 2

1) read X
2) compute X+1
3) write X

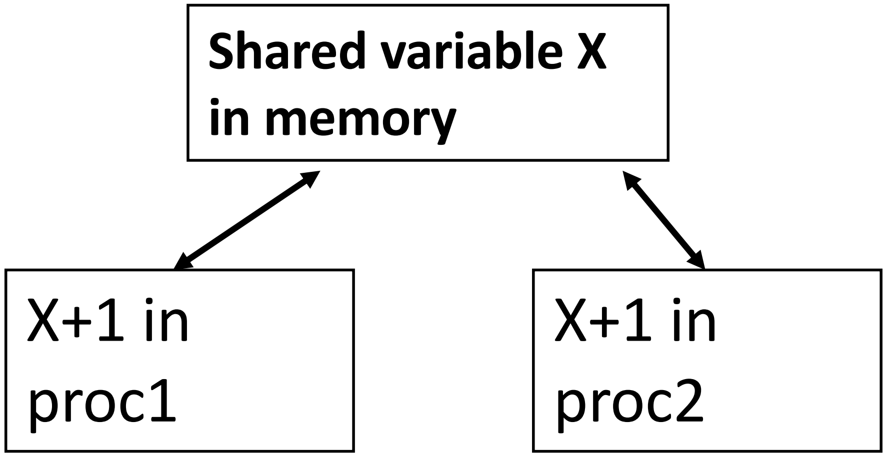


- Sequential execution



Accessing Shared Variables

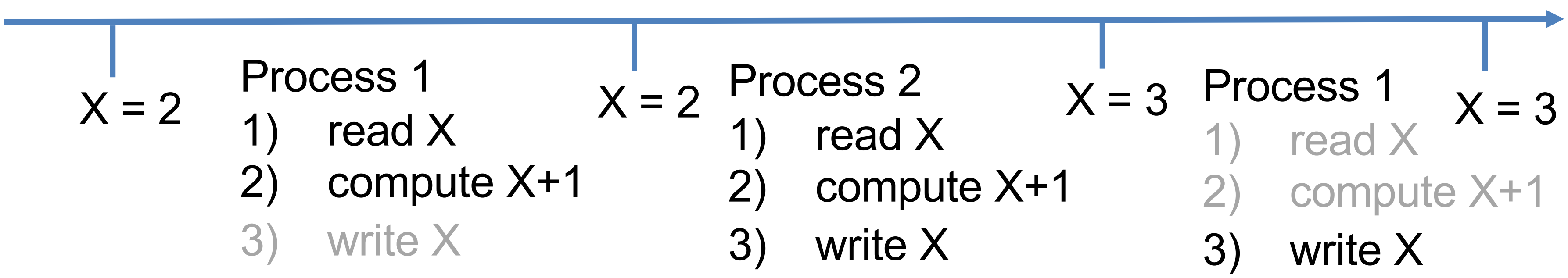
- If multiple processors want to write to a shared variable at the same time there may be conflicts :



Process 1 and 2

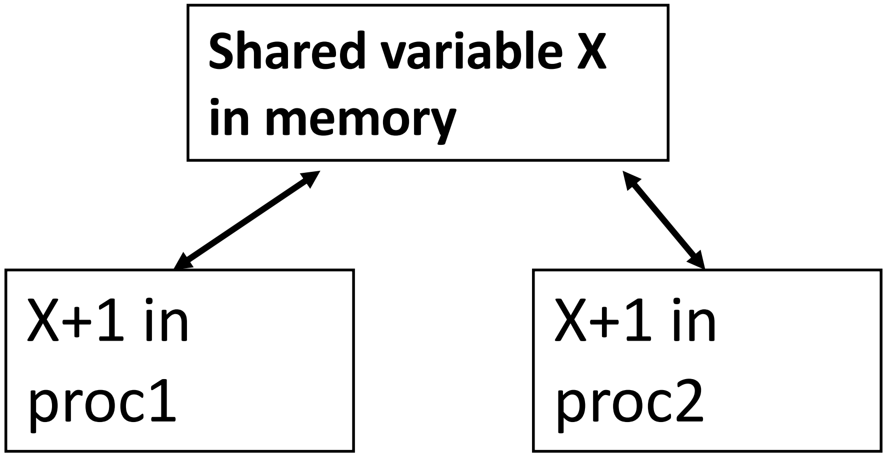
1) read X
2) compute X+1
3) write X

- Parallel execution



Accessing Shared Variables

- If multiple processors want to write to a shared variable at the same time there may be conflicts :



Process 1 and 2

1) read X
2) compute X+1
3) write X

- Programmer, language, and/or architecture must provide ways of resolving conflicts

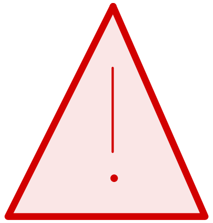
Accessing Shared Variables

- Race condition:
  - Application behavior depends on the sequence or timing of processes which should operate properly
  - (Critical) race conditions result in invalid execution and bugs (example before)

- Solution:
  - Lock: Mutual exclusive access to shared resources
    - Process 1 locks X
    - Until Process 1 unlocks X, nobody can even read X
    - Another source of overhead!

Accessing Shared Variables

- Lock introduces deadlock risk:
  - The processes P_A and P_B need two resources R_A and R_B
  - P_A obtains R_A
  - P_B obtains resource R_B
  - R_B is not available for P_A, so the process enters into a waiting state
  - R_A is not available for P_B, so the process enters into a waiting state
  - Both will be forever in waiting state!



- In general, deadlock arises when members of a **group of processes that holds resources** are **blocked indefinitely** from **access to resources held by other** processes **within the group**

Accessing Shared Variables

- Lock introduces deadlock risk:
  - The processes P_A and P_B need two resources R_A and R_B
  - P_A obtains R_A
  - P_B obtains resource R_B
  - R_B is not available for P_A, so the process enters into a waiting state
  - R_A is not available for P_B, so the process enters into a waiting state
  - Both will be forever in waiting state!

  - In g … **pro** … rom **acc** … **roup**

With the MPI we cover, we do **not** incur in **race conditions** but, possibly, we might introduce (and we should avoid!) **deadlocks**

Parallelization Example

- We now consider the following examples of parallelization:
  - Compute the sum of N numbers
- The example is provided in a C-like (non-existing) language
- We will consider a shared memory architecture

Sequential solution

```
#define N 100000
int a[N];
int i, s; //i: counter, s: sum
...
void main(){
     s=0;
     for (i=0; i<N; i++)
          s = s + a[i];
...
}
```

Parallel solution

```
#define N 100000
#define M (N/nproc)

share int a[n];
share int par_sum[nproc];
int i, s;
...
void main(){
...
        s=0;
        for (i = M*IDproc; i < M*(IDproc+1);
                s = s+a[i];
        par_sum[IDproc]=s;
```

Sum of partial results for each processor

These are local variables: each processor owns a private copy

IDproc is the ID of the processor and equals 0, 1, 2..

Each processor sums a fraction M of the numbers

There is still the problem to sum the partial values

Parallel solution: summing the partial results

- First solution : The first processor (IDproc == 0) does the final summation

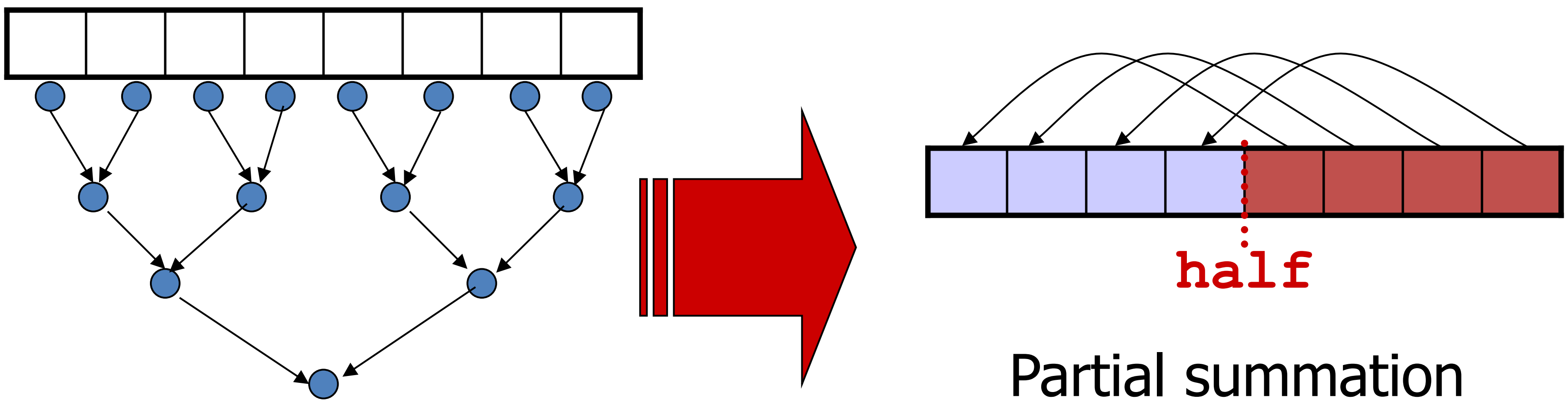
```
synch();

if (IDproc==0){
      for (i=1; i<nproc; i++)
            s=s+par_sum[i];
}
```

Waits that all the processors have ended their task

- Inefficient: this last step is not shared (serial component)

Parallel solution: parallelizing the last step

- Optimal solution: Partial sums are added in parallel by some of the processors (in $log_2 N$ steps)



$log_2 N$ steps instead of N

Partial summation is done iteratively

Parallel solution: parallelizing the last step

```
    int half= nproc/2;
    while(half>0){
        synch();
        if (IDproc<half)
            sum[IDproc]=sum[IDproc] +
                        sum[IDproc + half];
        half=half/2;
    }
}
```

Each leaf must be synchronized

References

- P. Pacheco, An Introduction to Parallel Programming, Chapters 1-2.

Additional References

- Hennessy, J. L. and Patterson, D. A. Computer Architecture: A Quantitative Approach.
- Patterson, D.A. and Hennessy, J.L., Computer Organization and Design: The Hardware/Software Interface.
- D. Dowd, High Performance Computing.
- D. Kuck, High Performance Computing. Oxford U. Press (New York) 1996.
- D. Culler and J. P. Singh, Parallel Computer Architecture.

Credits

- A. Majumdar. Introduction to Supercomputers, Architectures and High Performance Computing.
- J. M. Orduña Introduction to HPC technologies.
- A. DeConinck. High Performance Computing.
- J. Boisseau. Introduction to High Performance Computing: Parallel Computing, Distributed Computing, Grid Computing and More.
- T. Sterling. High Performance Computing: Models, Methods, & Means. An Introduction.