HPC : High Performance Computing

Master 2 SID 2018-2019

Benoist GASTON

Previously

HPC Overview

Performance Computing

- Hardware
    - Measurement: FLoating OPeration per second (FLOPs)
- How to increase hardware performance ?
    - Increase FLOPs per CPU (CPU faster)
    - Increase the number of CPUs: cumulate the performance

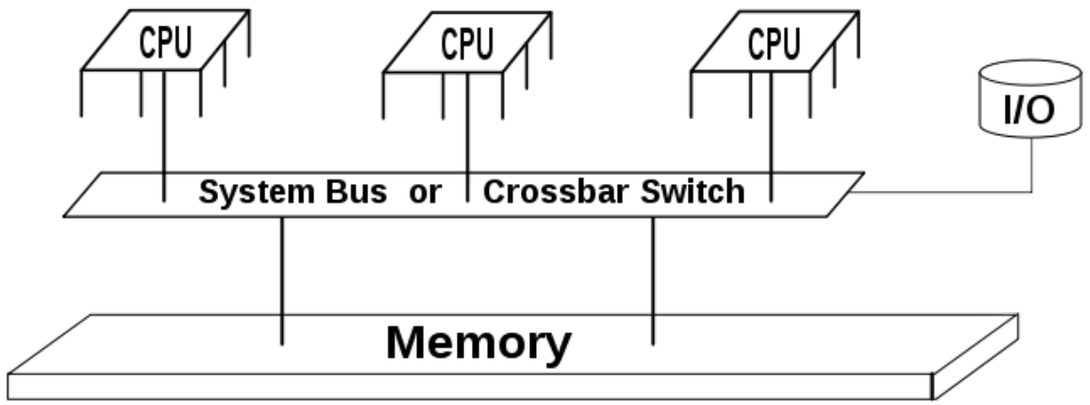
Performance Computing

- Software
  - Measurement: elapsed time from the begin to the end of an execution
  - How to increase software performance ?
    - Reduce the algorithmic complexity: use faster algorithm
    - Optimize the current algorithm for the hardware target

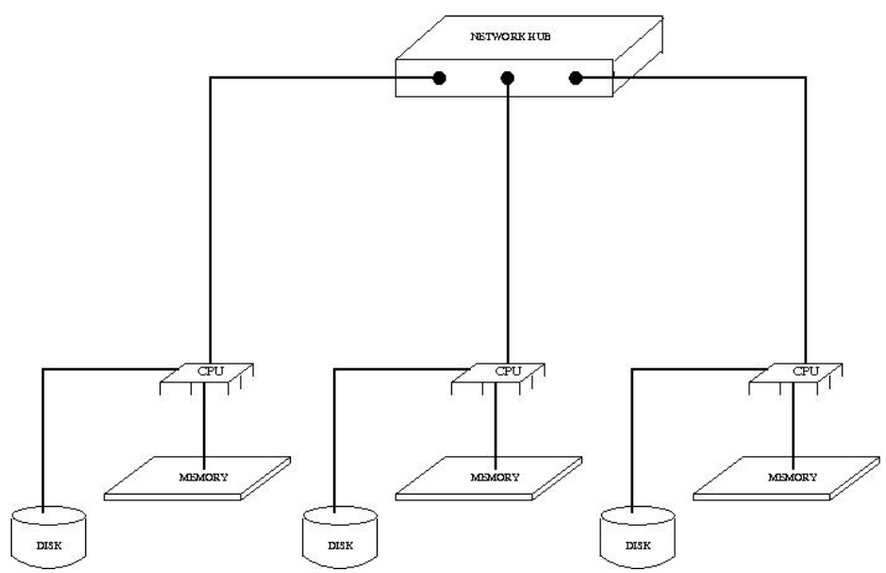
Parallel Computing

Increase the number of cpus (*supercomputer* and *parallelism*)

- **Shared memory:** many cpus share the same memory (RAM) (current architecture for PC)



- **Distributed memory:** each cpu has its own memory (RAM)



Increase the number of cpus (*supercomputer* and *parallelism*)

Benefit

- Accumulated performances (if the problem is designed for)
- with p cpus the code run p x faster

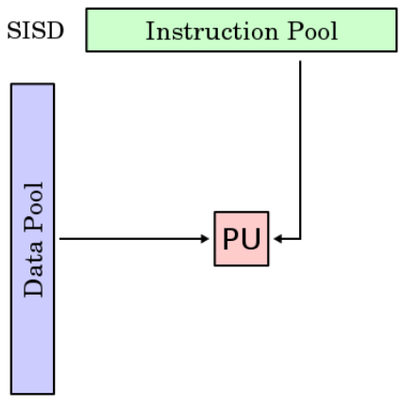
But

- Be careful: needs coordination and communication (network in case of distributed memory)

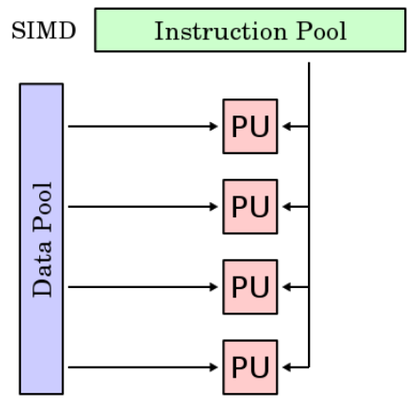
Parallel architectures and paradigms

Flynn's Taxonomy

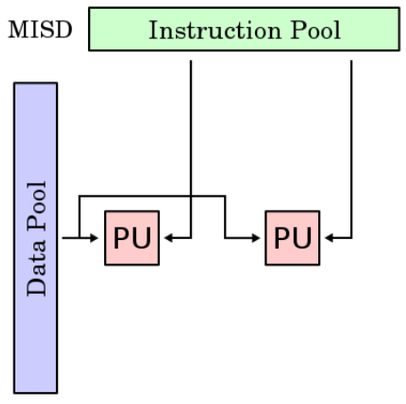
Single instruction single data (SISD)
Classic sequential computer



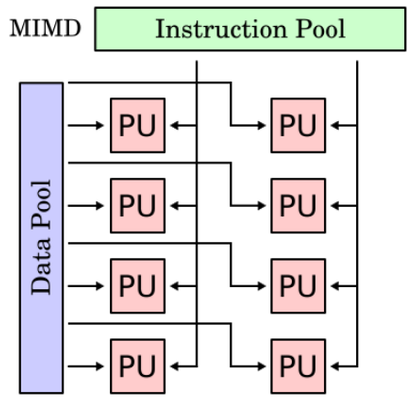
Single instruction, multiple data (SIMD)
Vectorization (ex. GPU)



Multiple instruction, single data (MISD)
Uncommon, used for fault tolerance



Multiple instruction, multiple data (MIMD)
Classic distributed tasks parallelism



Flynn’s Taxonomy Extended

- Single Program Multiple Data (SPMD)
  - A subcategory of MIMD
  - Multiple autonomous processors run the same program on multiple data
    - The most common type in massively parallel programs
- Multiple Program Multiple Data (MPMD)
  - A subcategory of MIMD
  - Multiple autonomous processors run independent programs on multiple data with direct or indirect communications (or not)
  - Typical exemple: coupling different software

Three main paradigms

Share the effort: vectorization

Distribute the effort: data parallelism

Assign effort on different tasks: task parallelism

SIMD: Vectorization

- Multiple processing elements perform the same operation on multiple data points simultaneously

```
a = [1,9,2,8]
b = [3,4,5,6]
for i in range(4):
  Result[i] = a[i] * b[i]
```

a=	1	9	2	8
b=	3	4	5	6
Result=	3	36	10	48
iteration	1	2	3	4

SISD

```
a = [1,9,2,8]
b = [3,4,5,6]
Result[1:4] = a[1:4] * b[1:4]
```

a=	1	9	2	8
b=	3	4	5	6
Result=	3	36	10	48
iteration	1			

SIMD

SIMD: Vectorization

```
#tabx and taby two arrays of size n
#res array of size n
for i in range(n) :
    res[i] = tabx[i] * taby[i]
```

Vectorizable:
same operation for any `i`

```
#tabx and taby two arrays of size n
#res array of size n
# cond is a function
for i in range(n) :
    if cond(tabx[i]) == A :
        res[i] = (taby[i] - 1) / tabx[i]
    elif cond(taby[i]) == B :
        res[i] = (tabx[i] - 1) / taby[i]
    else :
        res[i] = 1
```

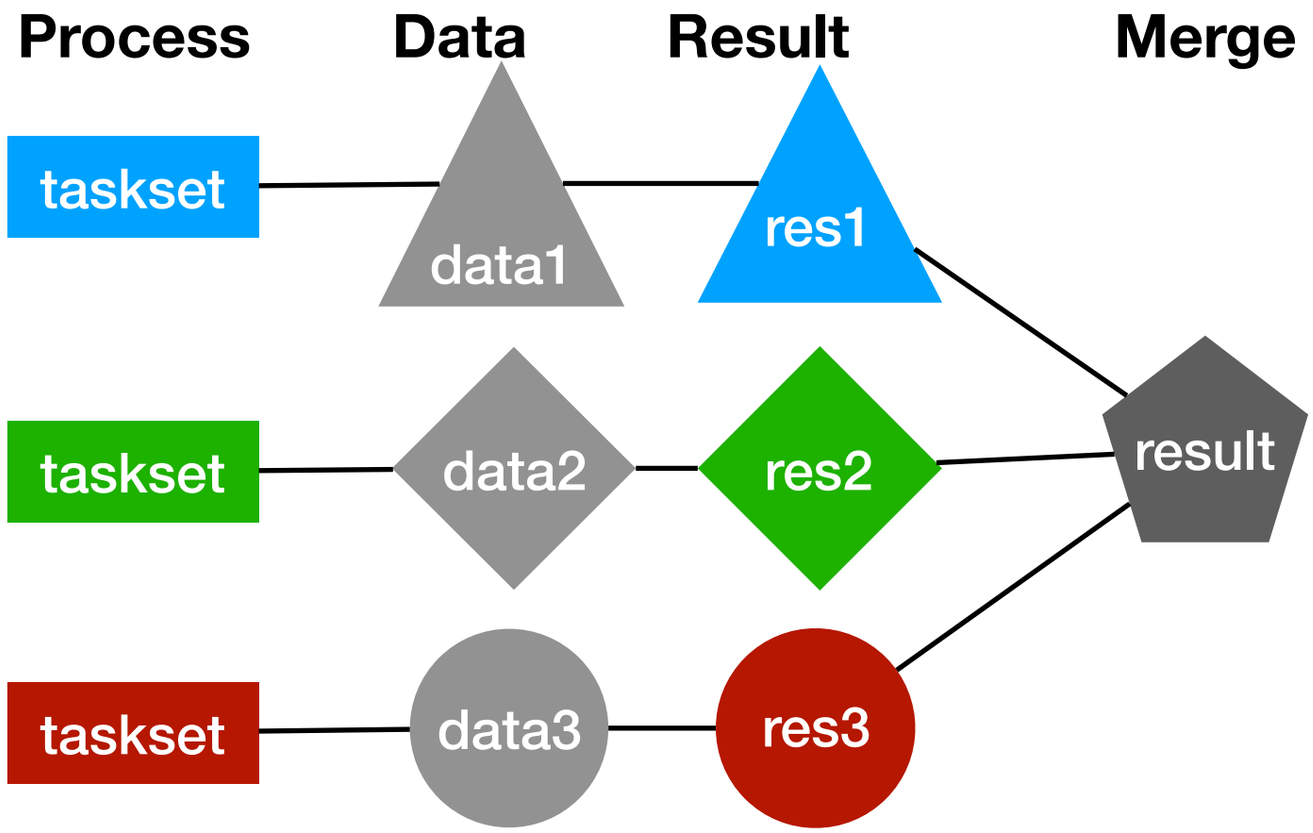
Not vectorizable:
for differents `i`, operations differ
(depend on condition)

```
#tabx and taby two arrays of size n
for i in range(n) :
    taby[i] = (taby[i - 1]) / tabx[i]
```

Not vectorizable:
the new value of `taby[i]`
depends on the new value of `taby[i-1]`
can not be modified at the same time

MIMD: data parallelism

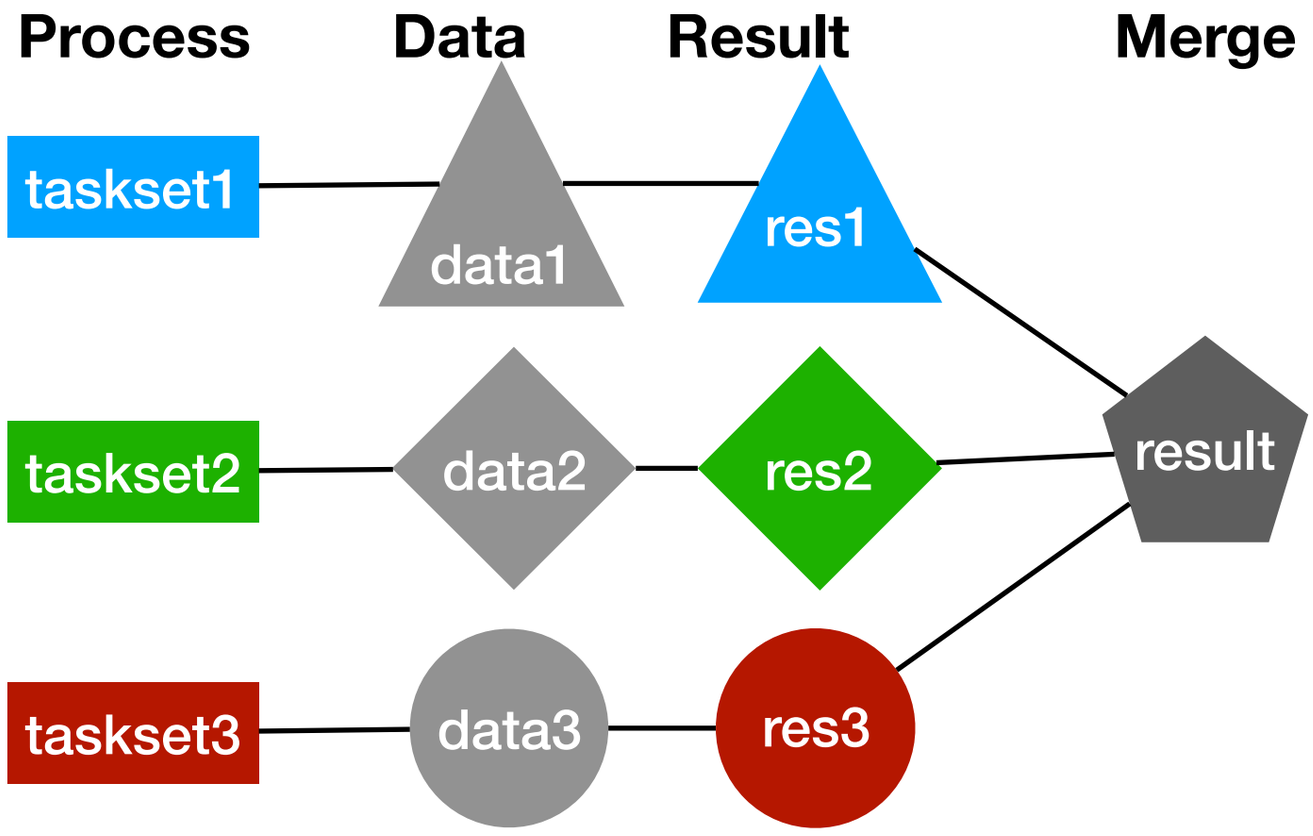
- Multiple processing elements perform the same set of multiple instructions on multiple data points simultaneously




```
#distribute data
if proc == 1:
    data = data1
if proc == 2:
    data = data2
if proc == 3:
    data = data3
#each process call the same taskset
res = taskset(data)
result = merge(res)
```

MIMD: task parallelism

- Multiple processing elements perform different sets of multiple instructions on multiple data points simultaneously




```
#assign tasks (and data)
if proc == 1:
     tasks = taskset1
     data = data1
if proc == 2:
     tasks = taskset2
     data = data2
if proc == 3:
     tasks = taskset3
     data = data3
#each process call its own taskset
res = taskset(data)
result = merge(res)
```

Communication

- Embarrassing parallelism
  - each task is independent: no shared data during the execution
- Except for embarrassing parallelism each parallel program needs data exchange between tasks
  - In shared memory paradigm, data is directly shared (single address space)
  - In distributed memory paradigm, data is shared by communication or remote access (RDMA: Remote Direct Memory Access)

Classes of parallel computers

- Shared memory
  - Multicore computer: one processor includes multiple processing units (standard in current PC)
  - Symmetric Multiprocessor (SMP): multiple identical processors sharing memory

Classes of parallel computers

- Distributed computers: multiple processors with not shared memory

  - Cluster computers: multiple computers connected by a local network

  - Grid: multiple computers connected by an internetwork, typically for embarrassingly parallel computing

  - Massively parallel computer (MPP): on computer with many processors networked, typically more than 100 processors with high-speed interconnect

Classes of parallel computers

- Specialized parallel computers: specifically design for one class of parallelism

    - FPGA: Chip than can be reconfigurate for a specific algorithms, used as a co-processor (for R&D, not necessary parallel)

    - ASIC: chip design for a specific algorithm (Embedded application, not necessary parallel)

    - Vector processor: computer than can execute the same instruction on large sets of data.

    - Graphics Processing Units (GPU): design for linear algebra matrix operation; its a kind of vector processor, used as a co-processor

Parallelism vs Concurrency

Parallelism:

- use multiple processors to make a computation faster

Concurrency:

- permit multiple tasks to proceed without waiting for each other

Different goals that share implementation aspects

- Scientific computing cares more about parallelism
- Concurrency is rarely needed

Parallel Programming

- Decomposition of the complete task into independent subtasks

- Distribution of the subtasks over the processors minimizing the total execution time

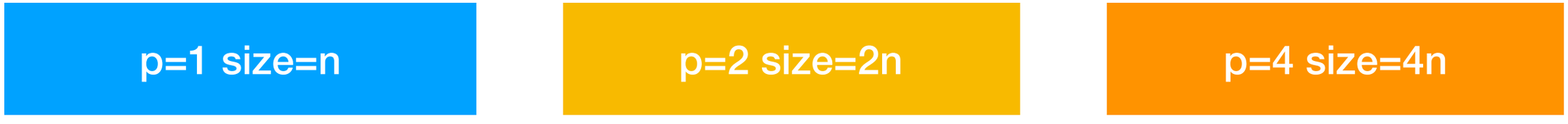
- synchronisation of the individual processes

Parallel Issues

Extra definitions

- **Speedup:** measures the relative performance of two systems processing the same problem: $S_p = \frac{T_1}{T_p}$, p = nb process
- **Scalability:** capability of a process to handle a growing amount of work
  - **Strong scalability:** elapsed time variation with the number of CPU for a fixed total problem size

  - **Weak scalability:** elapsed time variation with the number of CPU for a fixed problem size per CPU.



- **Efficiency:** rate between effective speedup S_p and theoretical speedup S_t (=p):

$$E_p = {}^{S_p}\!/_{S_t}$$

Parallel bound: Amdahl's Law

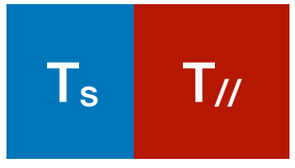
- Elapsed time of an application can be decompose into the parallelized time and the sequential time:

$$T_1 = T_s + T_{//}$$

p=1, $T_{//}$=90% T_1=10s

- The sequential part bound the total elapsed time on p proc:

$$T_p \geq T_s + \frac{T_{//}}{p}$$



p=10 T_p=2s

- Consequence: Speedup and Efficiency are bound:

$$S_p \leq \frac{p(T_s + T_{//})}{pT_s + T_{//}} \qquad E_p \leq \frac{T_s + T_{//}}{pT_s + T_{//}}$$

p=10, S_p=10/2=5, E_p=50%

Other parallel issues

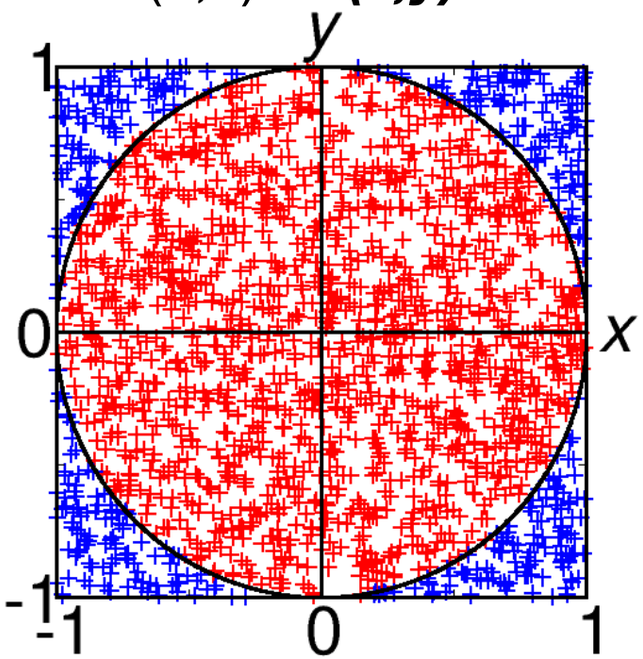
- **Overhead:** time spent in instruction added for parallel management (starting subtask, …)

- **Communication:** time spent in communication between process (network, memory access, message size, …)

- **Load unbalancing:** the amount of work is not equal for each process

Example:
Compute Pi with a Monte Carlo Method

- Monte Carlo: statistical method to solve a problem
- A circle C of radius r is inscribed inside a square $2r \times 2r$
- The ratio of the area of the circle to the area of the square is $\pi/4$
- consequence if you pick N points at random inside the square, approximately $M = {\pi N}/{4}$ of those points should fall inside the circle, i.e. $\pi = {4M}/{N}$
- for C of Radius ***r=1*** centered on (0,0) ***P=(x,y)*** is inside C if and only if $x^2 + y^2 \leq 1$



Pi with Monte Carlo

```
import numpy as np
import time
def picks(n):
    count_inside=0
    for i in range(n):
        x,y=np.random.random(2)*2-1
        if x*x+y*y <=1: count_inside+=1
return count_inside
```

Pi with Monte Carlo

- How to do it in parallel ?
- let each Process Unit do *n/npu* picks, where ***npu*** is the total number of PU : i.e. each PU launch **`picks(n/npu)`**
- Careful each PU compute a partial solution, needs to merge all partials solutions

Shared memory solution
python submit

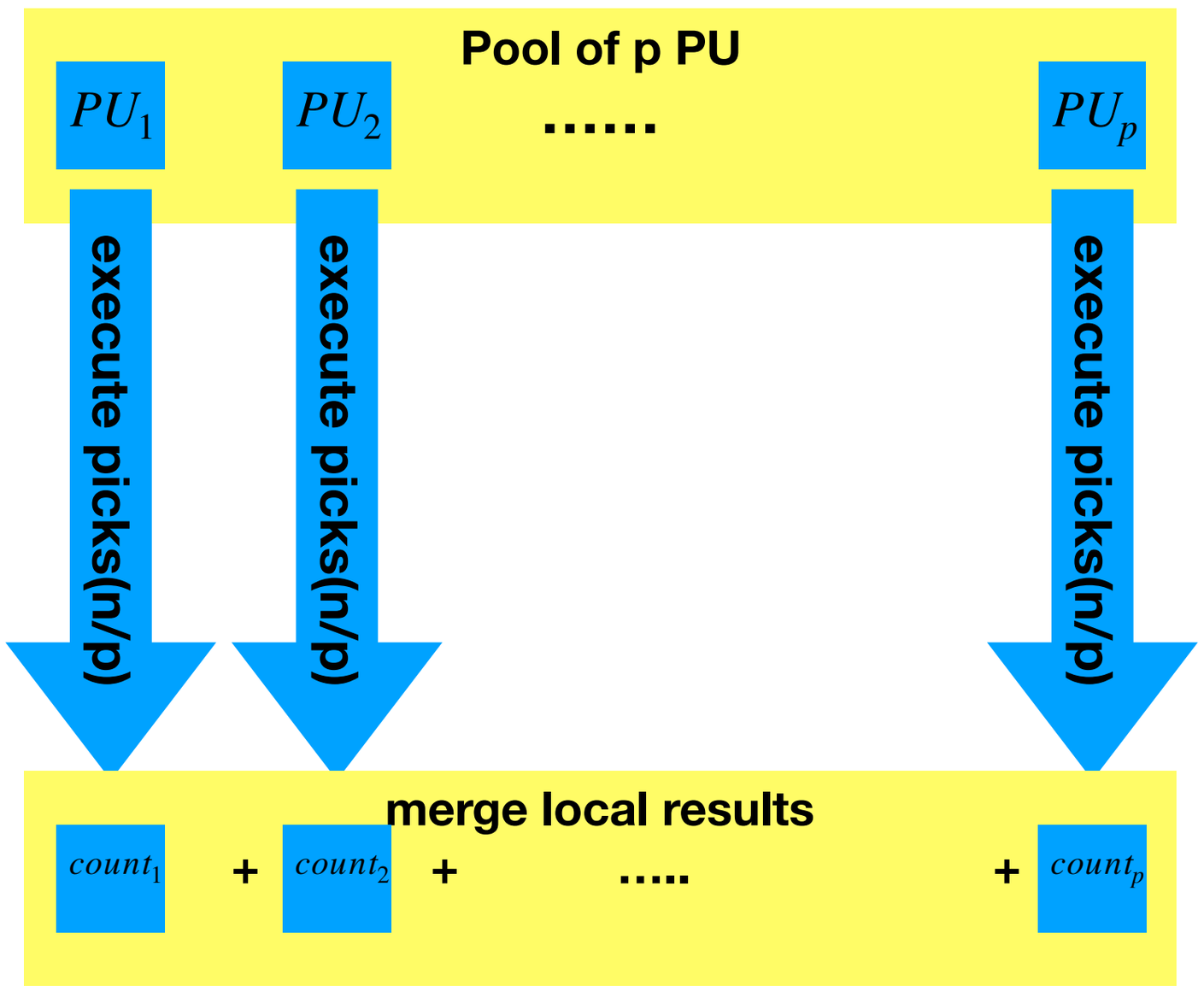
```
def par_picks(n,npu):
    from concurrent.futures import ProcessPoolExecutor
    pool = ProcessPoolExecutor(npu)
    nlocal=int(n/npu)
    futures = [pool.submit(picks,nlocal)] * npu
    results = [f.result() for f in futures]
    return results
```

Define a pool of npu PU

Let nlocal n/npu number of picks per PU

launch/submit npu instance of the picks function with nlocal

Parallel Monte Carlo PI



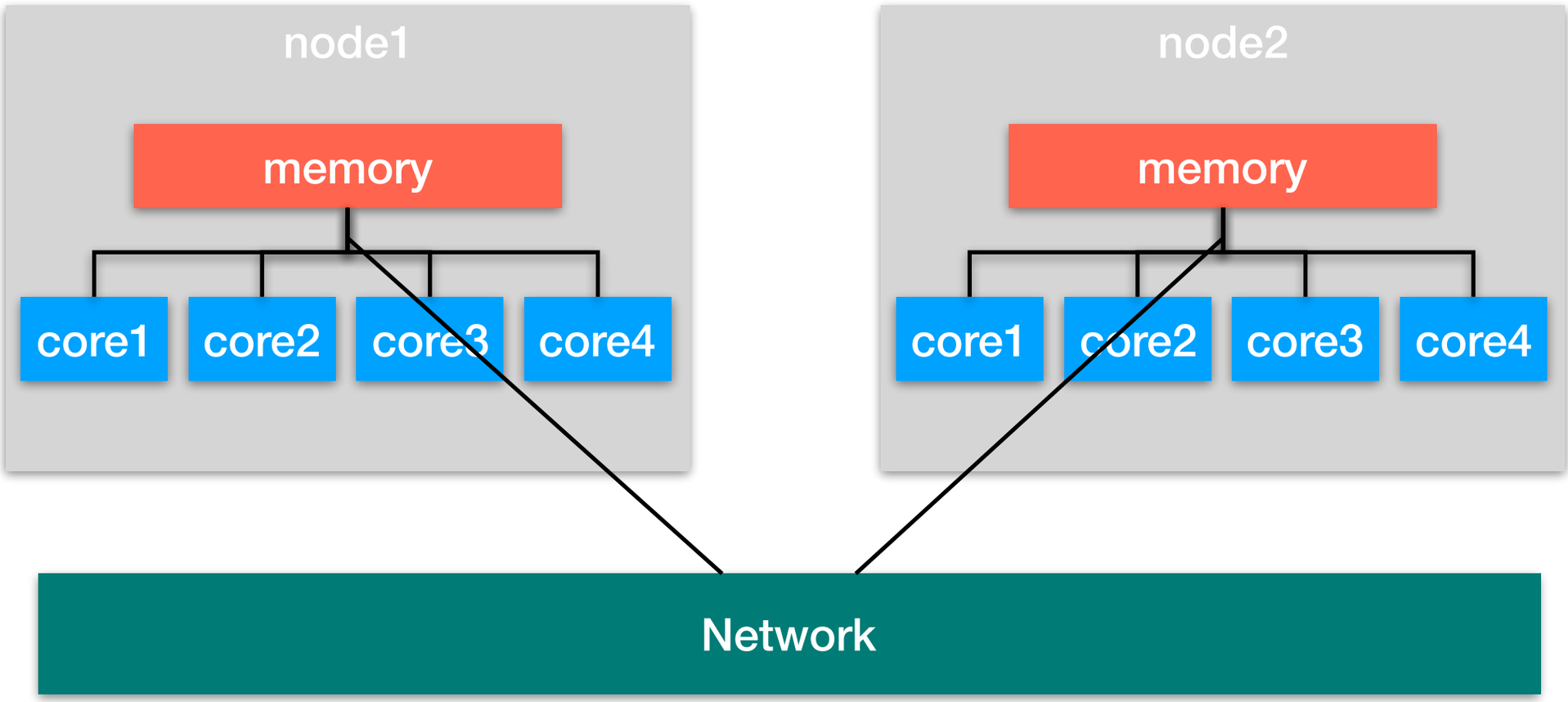
solution
Warning

nlocal: estimation, what append if nproc don’t divide n ?

```
def par_picks(n,nproc):
    from concurrent.futures import ProcessPoolExecutor
    pool = ProcessPoolExecutor(nproc)
    nlocal=int(n/nproc)
    futures = [pool.submit(picks,nlocal)] *nproc
    results = [f.result() for f in futures]
    return results
```

partial results, count_inside for each process: need to be merged (here a sum)

Distributed memory

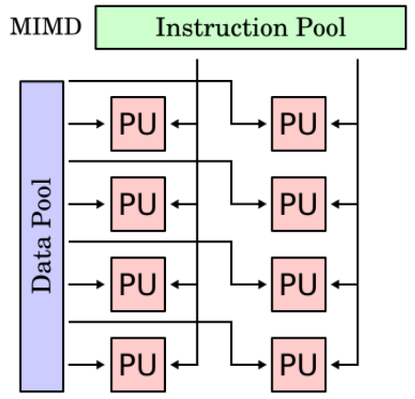


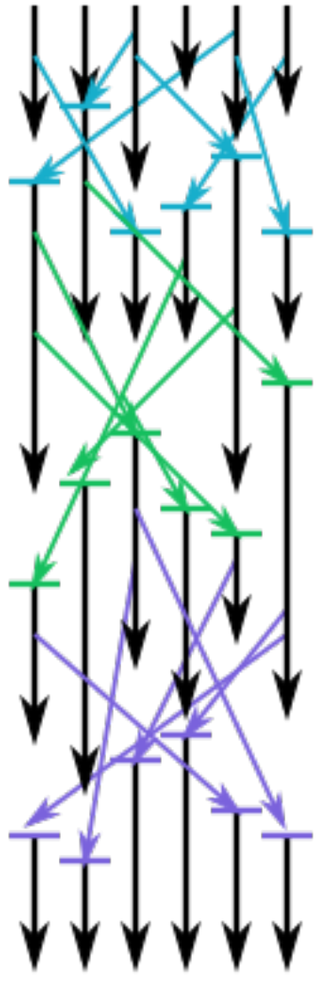
MPI:
Message Passing Interface

Principle

- multi-processing: n instances/ processes of the same program executed at the same time (MIMD:SPMD) on different computers

- each process communicate with other using message exchange

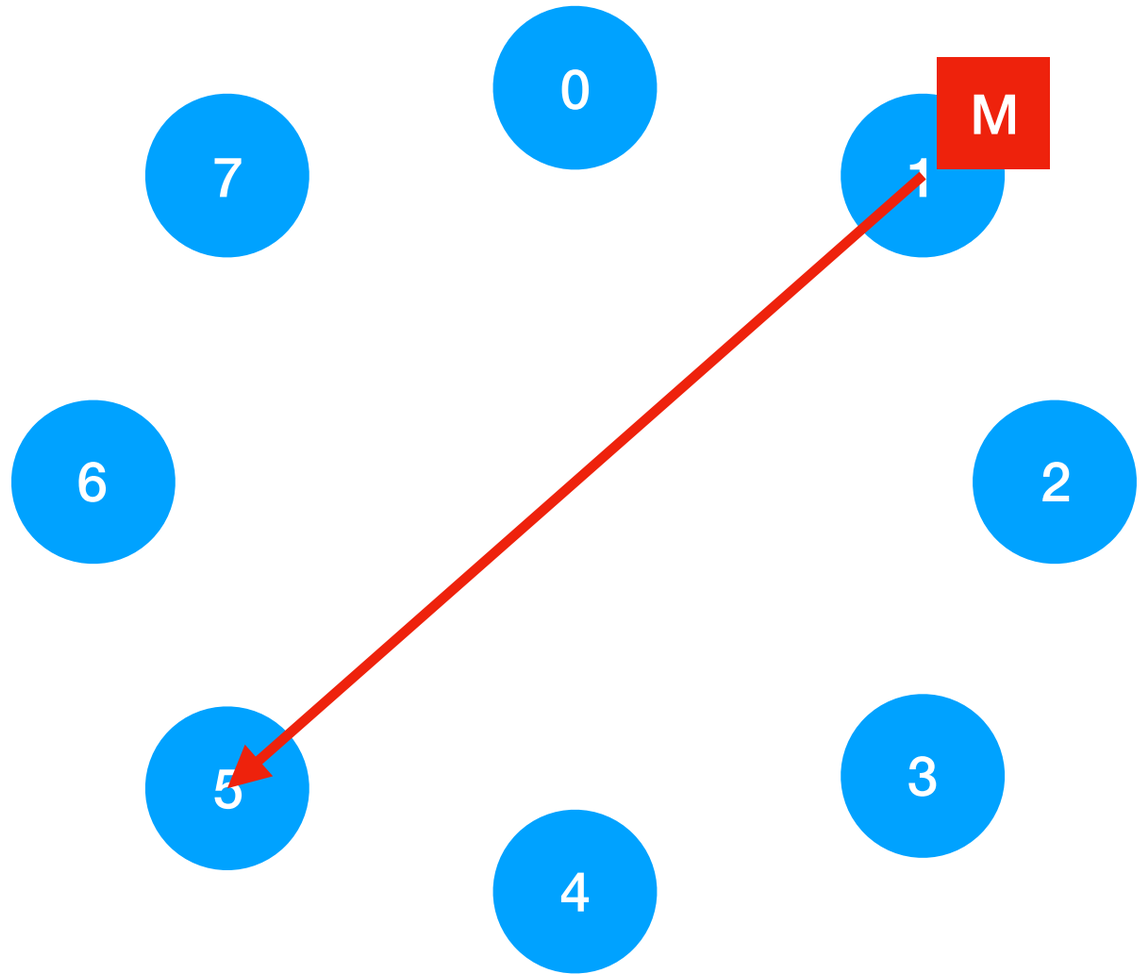




Glossary

- **Communicator** Group of processes that know each other and can communicate with each other
- Each process of a communicator is identify by his **rank** (from 0)
- **Communication** message exchange between two (or more) processes: on process send a message to another process who receive it

Scheme



Communicator of 8 processes

process of rank 1 send a message M to process of rank 5

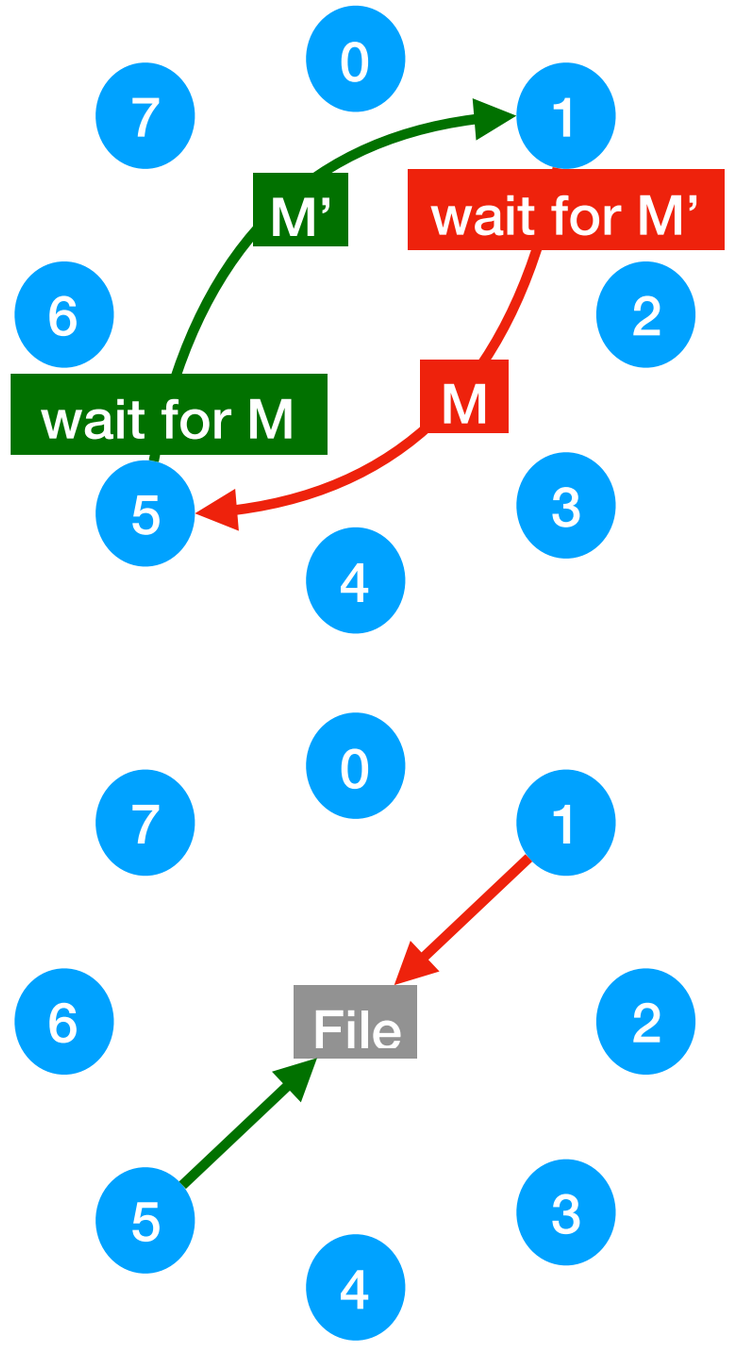
Communication/Synchronization Issues

Deadlock

- Two processes are waiting for each other to finish
- Usually caused by locks or blocking communication

Race condition

- two or more processes modify a shared resource (variable in shared memory, file, …)
- Result depends on which process comes first
- Can be avoided using locks but …
- … often cause deadlocks



Pi with Monte Carlo

```
def par_picks(n):
   comm=MPI.COMM_WORLD
   rank=comm.Get_rank()
   size=comm.Get_size()
   nlocal=int(n/size)
localcount=picks(nlocal)
count=localcount
if rank!=0:
   comm.send(localcount,dest=0)
if rank==0:
   for irank in range(1,size):
      rcount=comm.recv(source=irank)
      count+=rcount
return count
```

Decomposition Domain: Scheme and example

- **Domain**: representation of a part of a space (for example a mesh, a grid, or a picture, …) on which we have to manage data and ***computation***

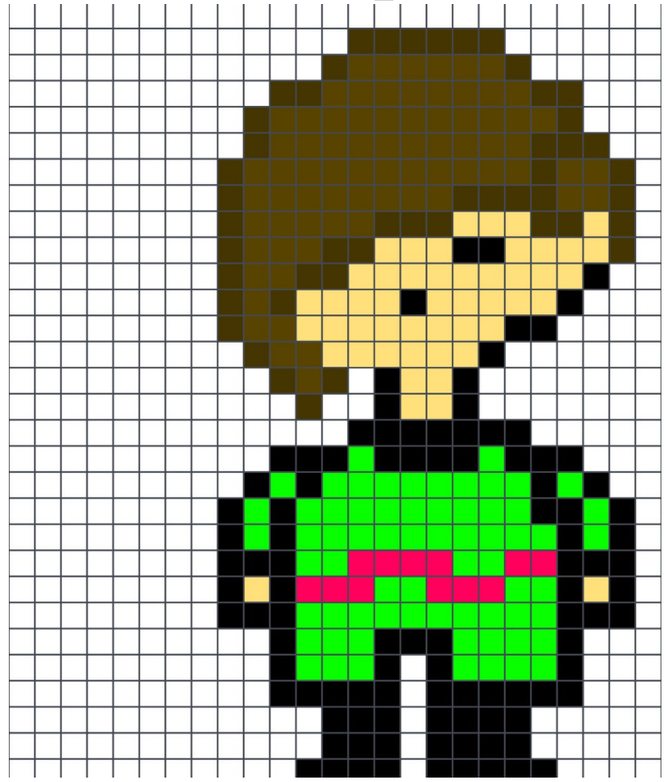
- Example an image domain is a 2D-array of pixels where data are the values of the pixels

- How to execute computation that in parallel?

- by **decomposition domain**:

    - the domain is split into subdomains

    - each process apply the compute kernel on one (or more) subdomain

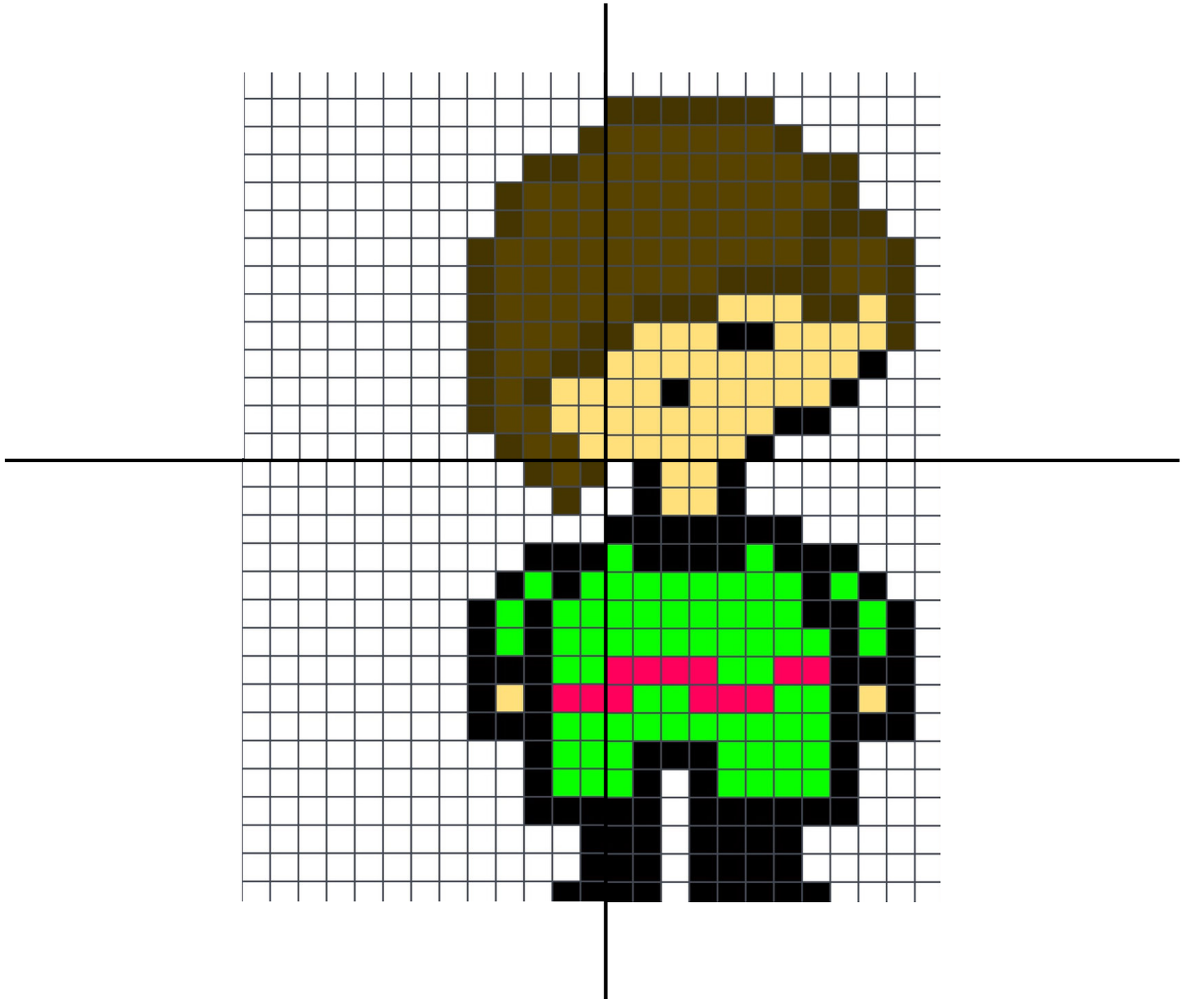


Domain: the grid
Data: the value of each pixel

Exemple with 4 processes

Process P0

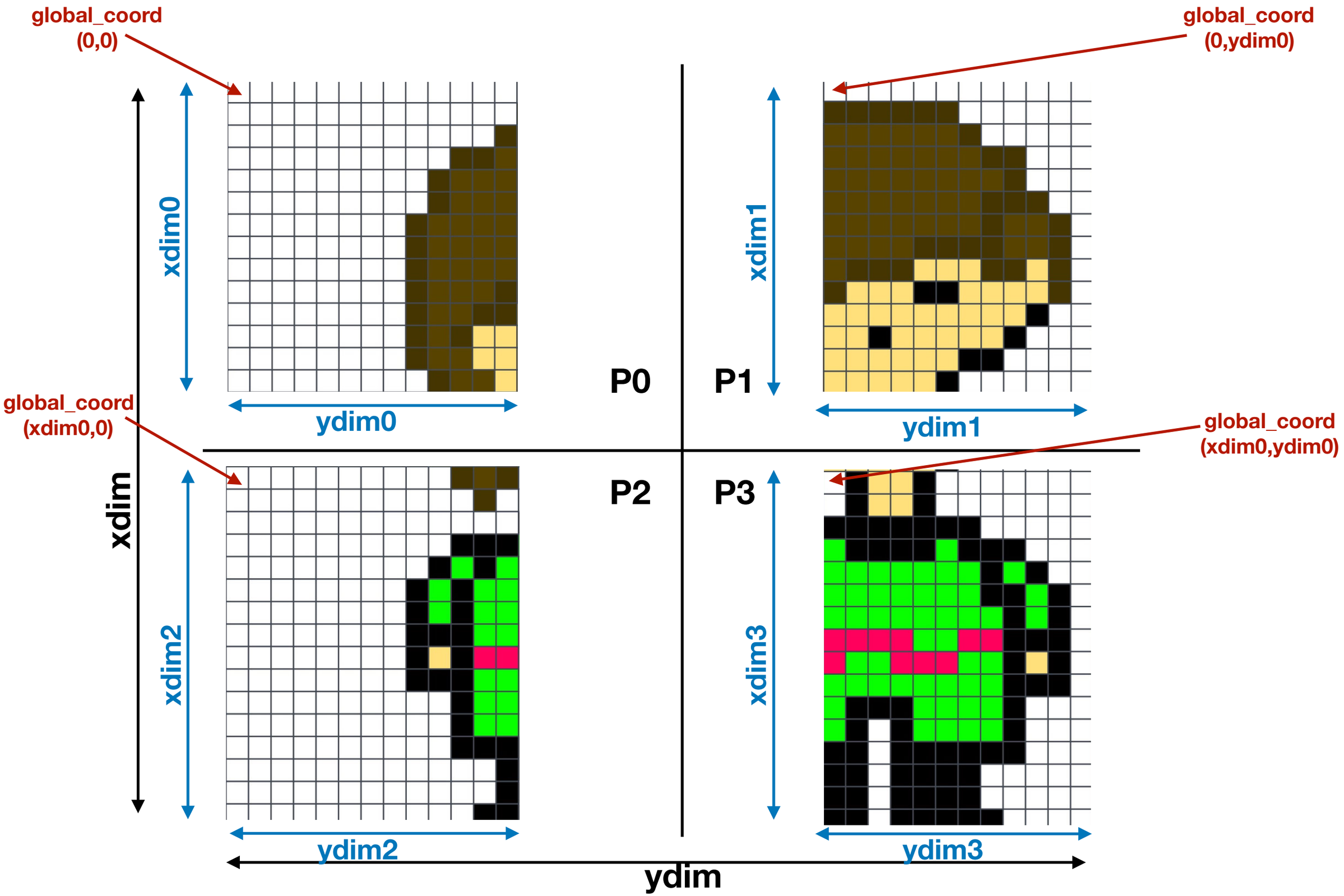
Process P1



Process P2

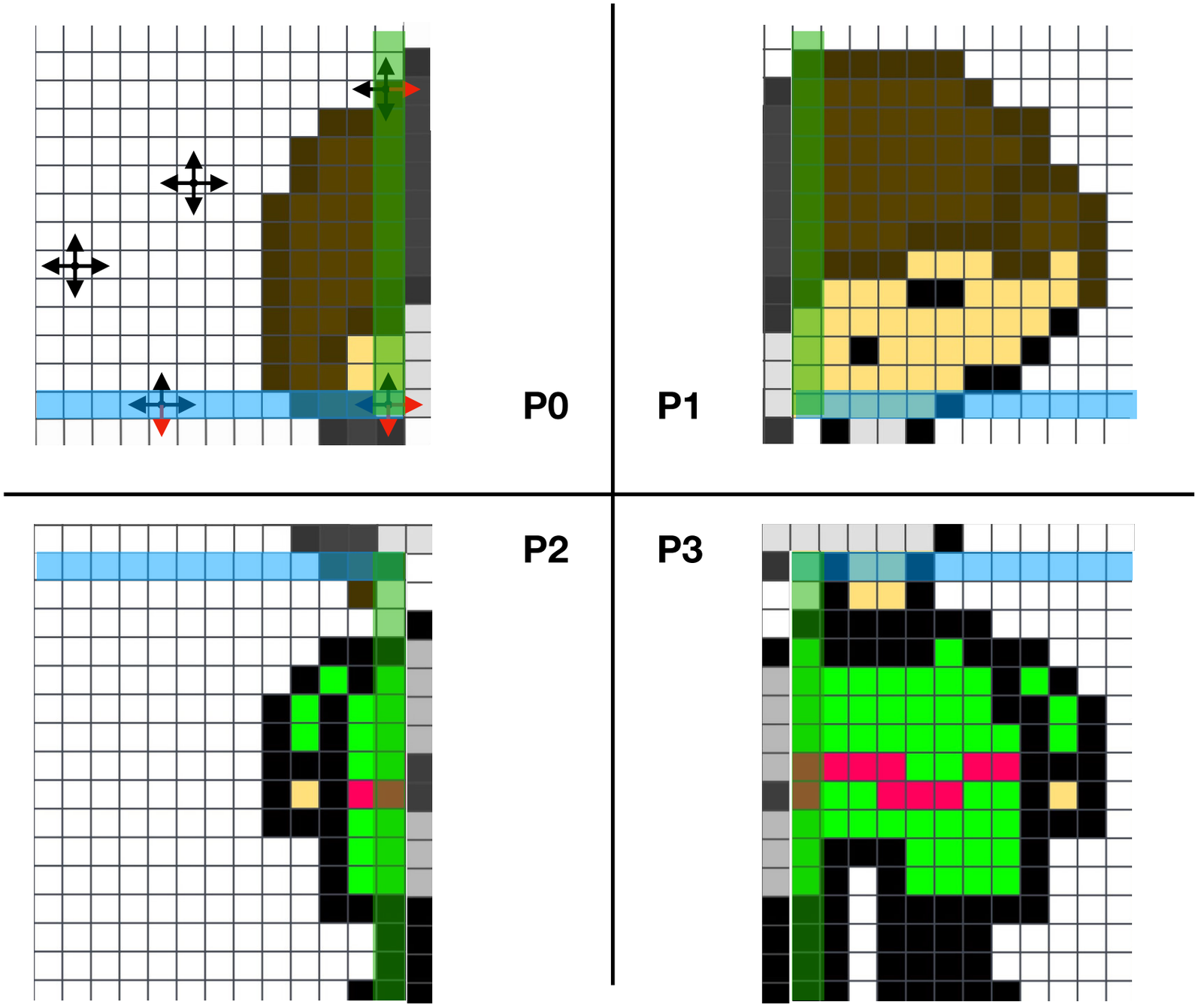
Process P3

Decomposition Domain: how to characterize local domain ?



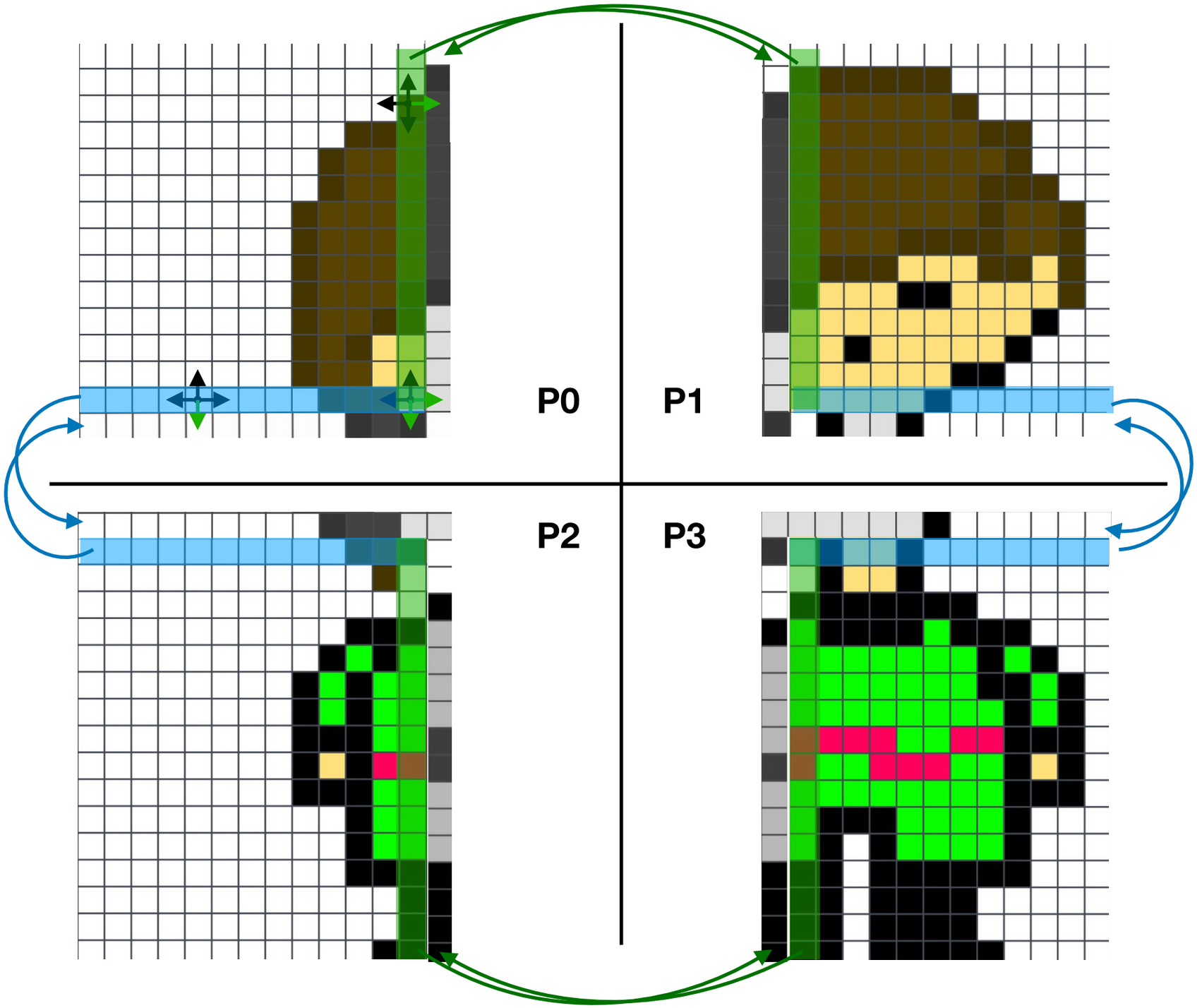
Decomposition domain: Most common issue

- Ghost and Interface



Decomposition domain: Most common issue

- Now we can communicate



Conclusion

Difficulties

- Correctness:
  - Verifying that subtasks are indeed independent
- Efficiency:
  - Attributing equal workloads to all processors
  - Taking into account computation and communication

Performance Optimisation

Performance Issues

Performance bounds

- **CPU bound** : process speed limited by CPU's speed (FLOPS) typically a task running a large amount of operations on few data

- **Cache bound** : process speed limited by the size and speed of cache, typically a task using more data than the cpu's register can store

- **Memory bound** : process speed limited by the size and speed of memory, typically a task using more data than the cache can store

- **I/O bound** : process speed limited by the I/O, typically read/write files

- I/O Bound < Memory Bound < Cache Bound < CPU Bound

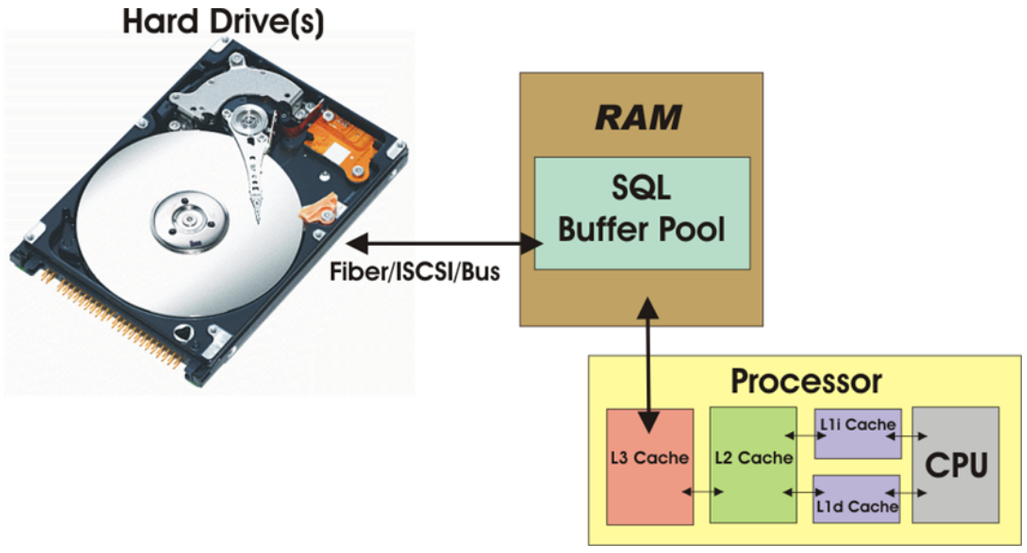
Consequences

- not only FLOPS that Matter
- Often data flow is necessary to « feed » the cpu

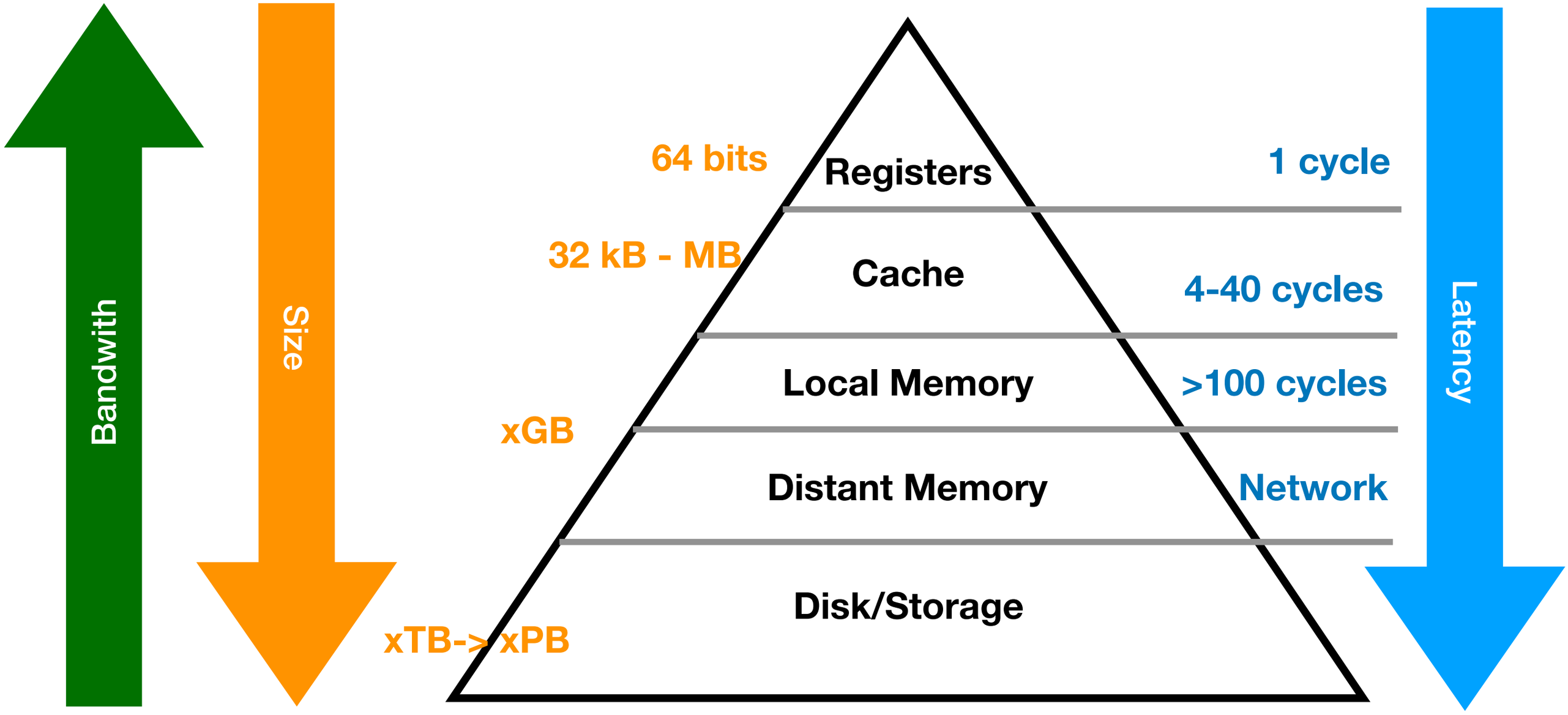
Extra Definitions

- **Latency**: elapsed time for access data, time between sending and receiving data; measure in second
- **Bandwith**: rate of data transfert, flow measure in byte/s

Memory Hierarchy



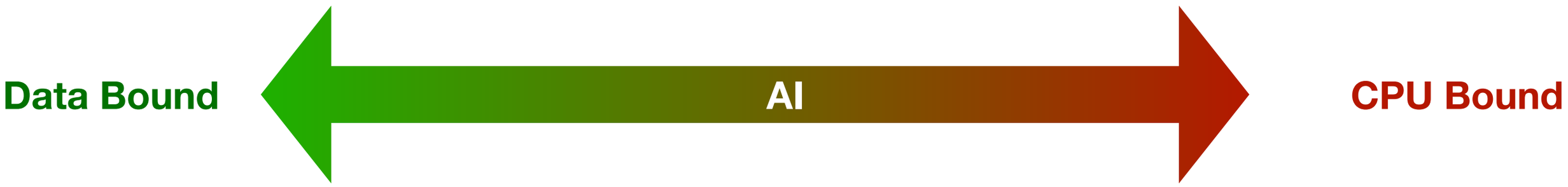
Memory Hierarchy



Roofline Model

- In two word : your application is CPU Bound or Data Bound

- Arithmetic Intensity
  - FLOPs : Floating point operation per second
  - MEMOPs : Memory Operation per second
  - AI=FLOPs/MEMOPs(=FLOP/MEMOP)



Arithmetic Intensity

```
for i in range(xdim) :
    for j in range(ydim) :
        x = tab1[i]
        y = tab2[j]
        res = (x * x + x * y + (y + x) * y)\
              * (y * y + y * x + ( x + y ) * x)
```

Floating Point Operations:

$13 \times ydim \times xdim$

Memory Operations :

$5 \times ydim \times xdim$

AI: 13/5

```
for i in range(xdim) :
    x = tab1[i]
    for j in range(ydim) :
        y = tab2[j]
        res = (x * x + x * y + (y + x) * y)\
              * (y * y + y * x + ( x + y ) * x)
```

Floating Point Operations:

$13 \times ydim \times xdim$

Memory Operations :

$(3 \times ydim + 2) \times xdim$

AI: 13 ydim /(3 ydim + 2)~13/3

```
for i in range(xdim) :
    for j in range(ydim) :
        res = tab[i - 1,j] + tab[i,j]\
              + tab[i + 1,j] + tab[i,j - 1]\
              + tab[i,j + 1]
```

Floating Point Operations:

$8 \times ydim \times xdim$

Memory Operations :

$6 \times ydim \times xdim$

AI: 8/6=4/3

Performance solutions

Performance solutions

- If application is CPU Bound:
- Reduce number of arithmetic operations
- Store rather than compute

```
for (i = 1; i < nx; i++) {
  y[i] = x[i]/h;
}
```

```
one_h = 1.0/h;
for (i = 1; i < nx; i++) {
  y[i] = x[i]*one_h;
}
```

```
for (i = 1; i < nx; i++) {
  for (j = 1; j < ny ; j++){
    val = 0 ;
    for (ii=i-1;ii<i+2;ii++)
      for (jj=j-1;jj<j+2;jj++)
        val+=tab[ii][jj];
  }
}
```

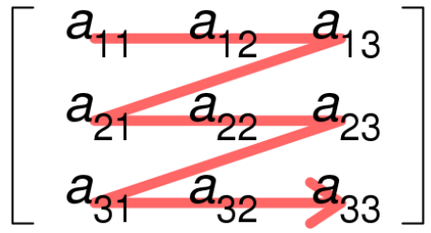
```
for (i = 1; i < nx; i++) {
  for (ii=i-1;ii<i+2;ii++)
    for(jj=0;jj<=2;jj++)
      c[jj]=tab[ii][jj];
  val=c[0]+c[1]+c[2];
  for (j = 2; j < ny ; j++){
    nc = 0;
    for (ii=i-1;ii<i+2;ii++)
      nc +=tab[ii][jj];

c[0]=c[1];c[1]=c[2];c[2]=nc;
    val=c[0]+c[1]+c[2];
  }
}
```

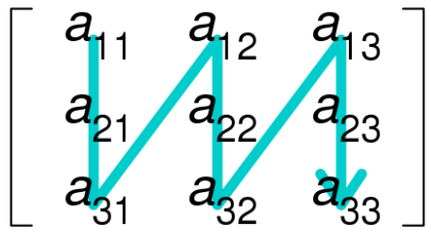
Performance solutions

- If application is Data Bound:
  - Reduce the number of data
  - compute rather than store
- in case of multidimensional arrays
  - be aware of RMO or CMO access
  - use cache blocking (submatrix)

```
temp_x = new int[n];
temp_y = new int[n];
temp_z = new int[n];

for (i=0;i<n;i++)
  temp_x[i]=do_this(coord_x[i]);
do_that(…,&temp_x);
for (i=0;i<n;i++)
  temp_y[i]=do_this(coord_y[i]);
do_that(…,&temp_y);
for (i=0;i<n;i++)
  temp_z[i]=do_this(coord_z[i]);
do_that(…,&temp_z);
```

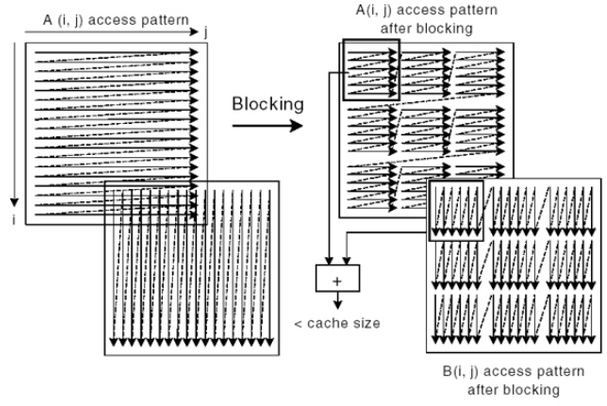
```
temp = new int[n];

for (i=0;i<n;i++)
  temp[i]=do_this(coord_x[i]);
do_that(…,&temp);
for (i=0;i<n;i++)
  temp[i]=do_this(coord_y[i]);
do_that(…,&temp);
for (i=0;i<n;i++)
  temp[i]=do_this(coord_z[i]);
do_that(…,&temp);
```





a11	a12	a13	a21	a22	a23	a31	a32	a33

a11	a21	a31	a12	a22	a32	a13	a23	a33



Performance Tools

Definitions

- **Profiling**: Analyse the execution of a code in order to know its behavior
- **Profile**: statistics about events observed during the execution of a code
- **Trace**: stream of recorded events during the execution
- **Profiler**: Tools for profiling
- **Hot Spot**: region of a code where a high proportion of executed instruction occur and/or where most time is spent during the program’s execution
- **Call Graph**: call-chains of the functions of a code during execution
- **Sampling**: statistical approach for profiling, the profiler probes the target code’s call stack at regular intervals not accurate but low overhead
- **Instrumentation**: collect approach for profiling, the profiler add instructions to the target program to collect required information (typically timers). Accurate but important overhead

Profile

Usage

- Identify bottleneck in performance
- Identify memory consumption issue (memory leak)

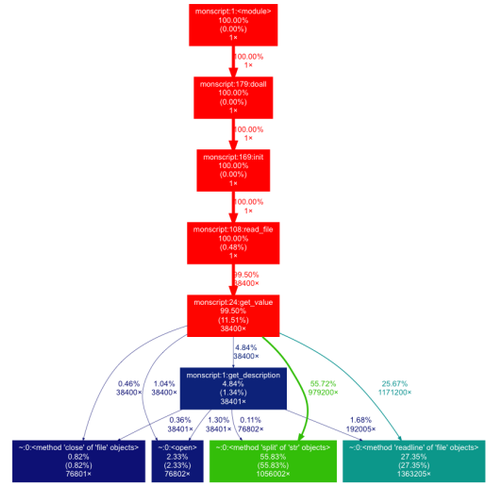
Flat Profile

- Statistical summary of the events observed
- information about time spent in each function and number of calls (by function)
- the grain of the information depending of the profiler:
  - functions code
  - code line
  - PU operations (assembly level)

Call graph profile

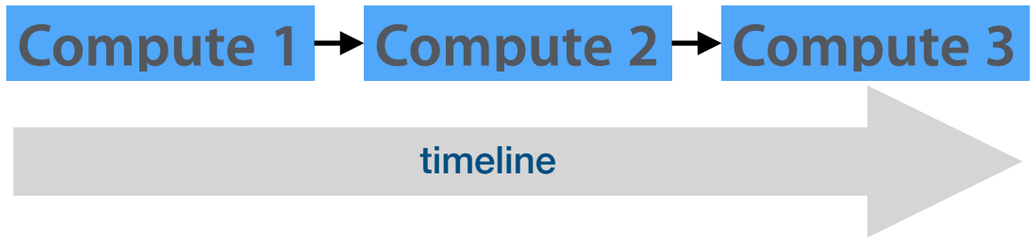
- shows how much time was spent in each function and its children

```
bash-3.2$ python -m cProfile monscript.py
         3744228 function calls in 29.798 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   29.798   29.798 monscript.py:1(<module>)
    38401    0.404    0.000    1.464    0.000 monscript.py:1(get_description)
        1    0.147    0.147   29.798   29.798 monscript.py:108(read_file)
        1    0.000    0.000    0.000    0.000 monscript.py:134(kernel)
        1    0.000    0.000   29.798   29.798 monscript.py:169(init)
        1    0.000    0.000   29.798   29.798 monscript.py:179(doall)
    38400    3.438    0.000   29.644    0.001 monscript.py:24(get_value)
   979202    0.078    0.000    0.078    0.000 {len}
    38553    0.007    0.000    0.007    0.000 {method 'append' of 'list' objects}
    76801    0.250    0.000    0.250    0.000 {method 'close' of 'file' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
  1363205    8.236    0.000    8.236    0.000 {method 'readline' of 'file' objects}
  1056002   16.511    0.000   16.511    0.000 {method 'split' of 'str' objects}
    76802    0.017    0.000    0.017    0.000 {method 'strip' of 'str' objects}
    76802    0.709    0.000    0.709    0.000 {open}
       54    0.000    0.000    0.000    0.000 {range}
```



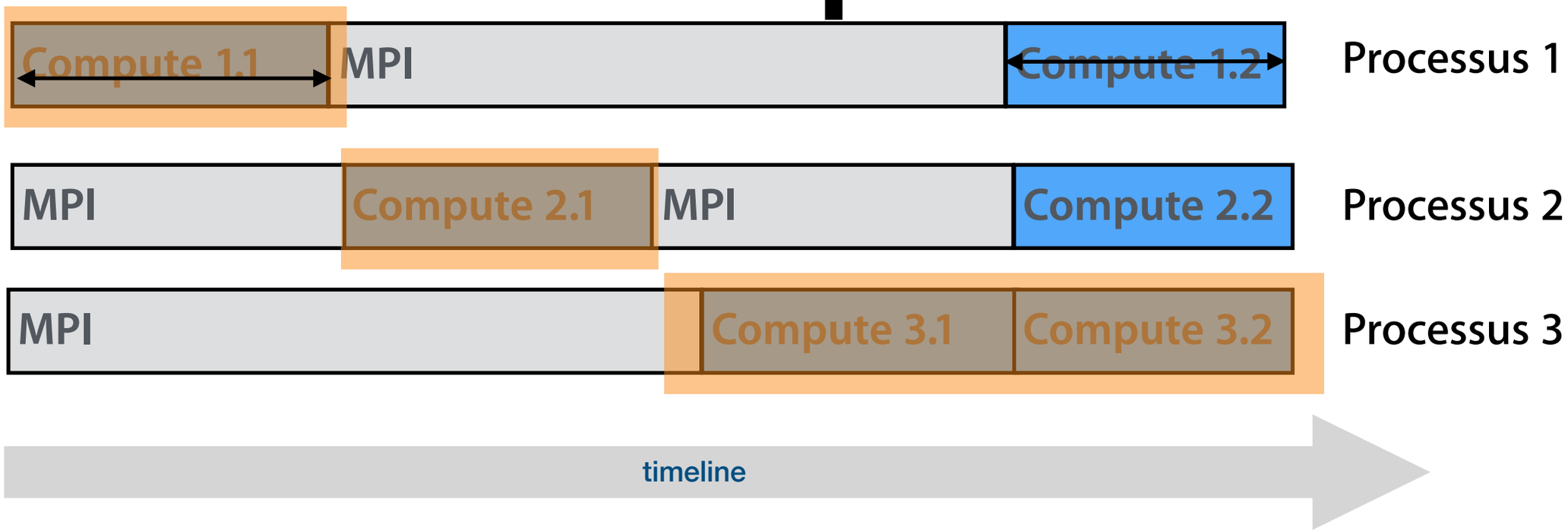
Trace

- Trace is the story of the execution of a program



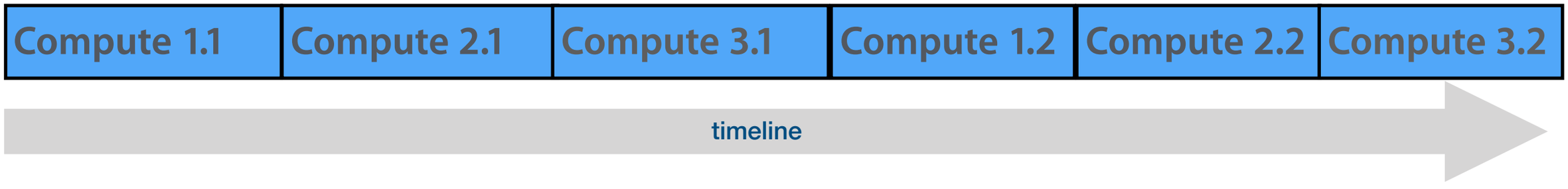
- each targeted event is recorded each time it occurs
- specially use for debugging and advanced performance analyse
- Typically trace of MPI events during a execution

Example



Good load balancing? **yes**

Effective parallelization? **no**



Multithreading and Task parallelism

Multithreading Vs Multiprocessing

Multiprocessing

- Several processes using separate memory space
- Advantages
    - Distributed memory => easy to increase memory
    - Code usually straightforward (coarse granularity)
    - A must for CPU Bound applications
- Disadvantages
    - Inter-process communication (IPC) little bit complicated with overhead
    - Large memory footprint

Multithreading

- Several threads (sub-processes of a parent process) using shared memory
- Advantages
    - Low memory footprint
    - Shared memory => easy to access to state from another context
    - Great for data bound applications
- Disadvantages
    - Difficult to interrupt/kill threads
    - Hard to schedule (execution, synchronization, race condition, …)
    - Easy to make mistakes, code difficult to understand

Multithreaded parallelization

- Multithreaded parallelization has existed for long time at certain manufacturer (CRAY, NEC, IBM, ..) but without standard languages

- The resurgence of shared memory multiprocessor need multithreaded parallelization to obtain performance and reduce memory footprint

- Difficult to use it with certain languages, especially python due to the GIL

- Shared memory particularly adapted to task parallelization and data flow approach

OpenMP

Introduction

- OpenMP: parallel programming model for shared memory architectures
- 28th October 1997: OpenMP massively adopted as an industrial standard
- Today: OpenMP also targets accelerators (i.e. co-processor like GPU), integrated systems and real-time systems
- Calculation tasks can access a common memory space => limits data redundancy and simplifies information exchanges
- Parallelization baser on light-weight processes (threads)
- OpenMP specification belong to the ARB (Architecture Review Board), only organisation responsible for its development

openMP Specification

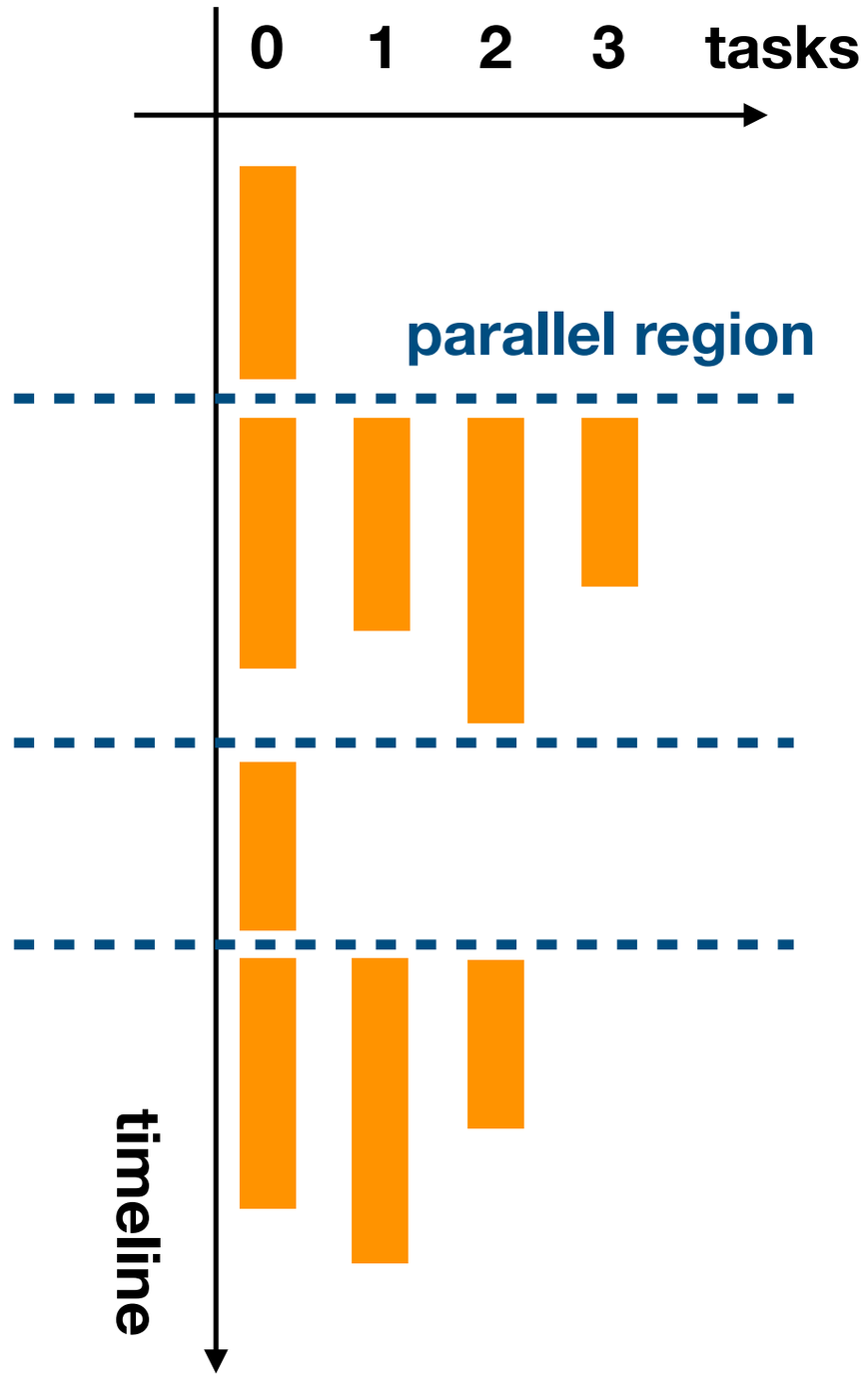
- OpenMP 1.0: October, 1997 (October 1998 for C/C++)
- OpenMP 2.0: November 2000
- OpenMP 3.0: May 2008, introduced the concept of tasks
- OpenMP 4.0: July 2013, introduced accelerator support, enhanced task parallelism

Definitions

- **Thread**: an execution entity with local memory (stack)
- **Team**: a set of one or several threads participating at the same parallel region
- **Task**: instance of executable code and associated data
- **Shared variable**: a variable whose the content is shared by all the task of a team (same name; same memory space)
- **Private variable**: a variable whose the content is specific for each task (same name; different memory space)

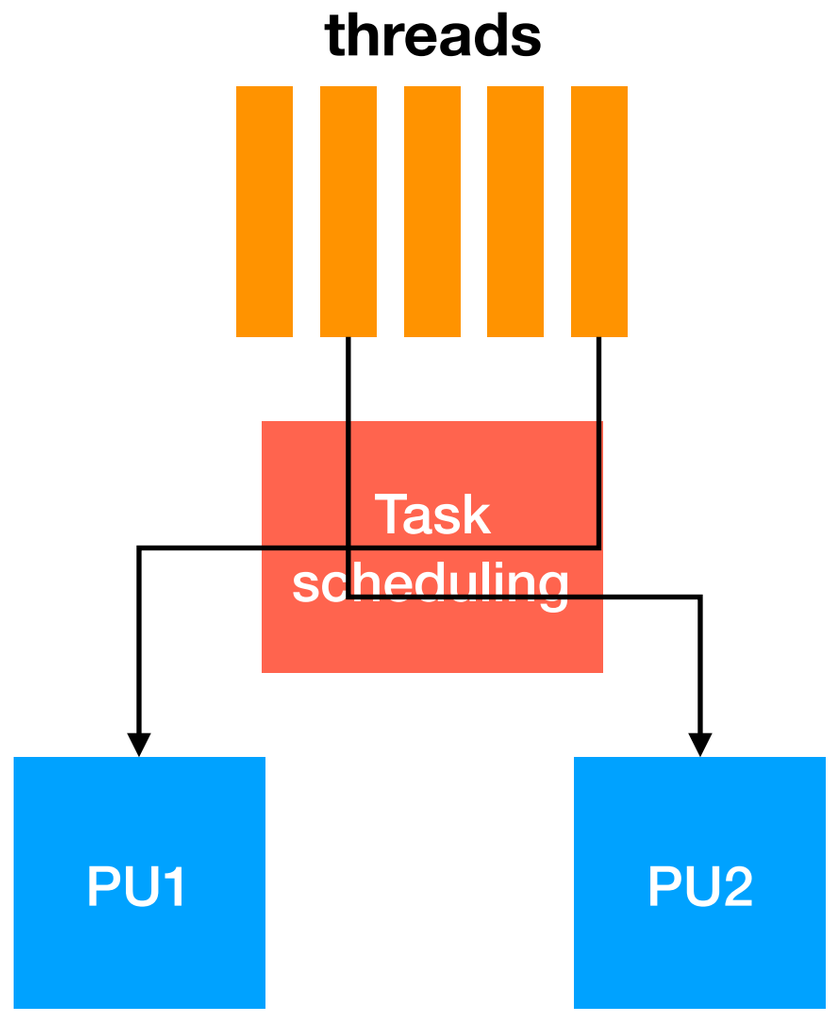
Concepts: execution model

- An openMP program begins as a sequential program: only one process
- OpenMP allows defining **parallel regions** which are code portions executed in parallel
- At the **begin** of a parallel region, new **threads** (and **implicit tasks**) are created
- During a parallel session **each thread executes its implicit task** concurrently with the others
- An OpenMP program consists of alternate sequential an parallel regions



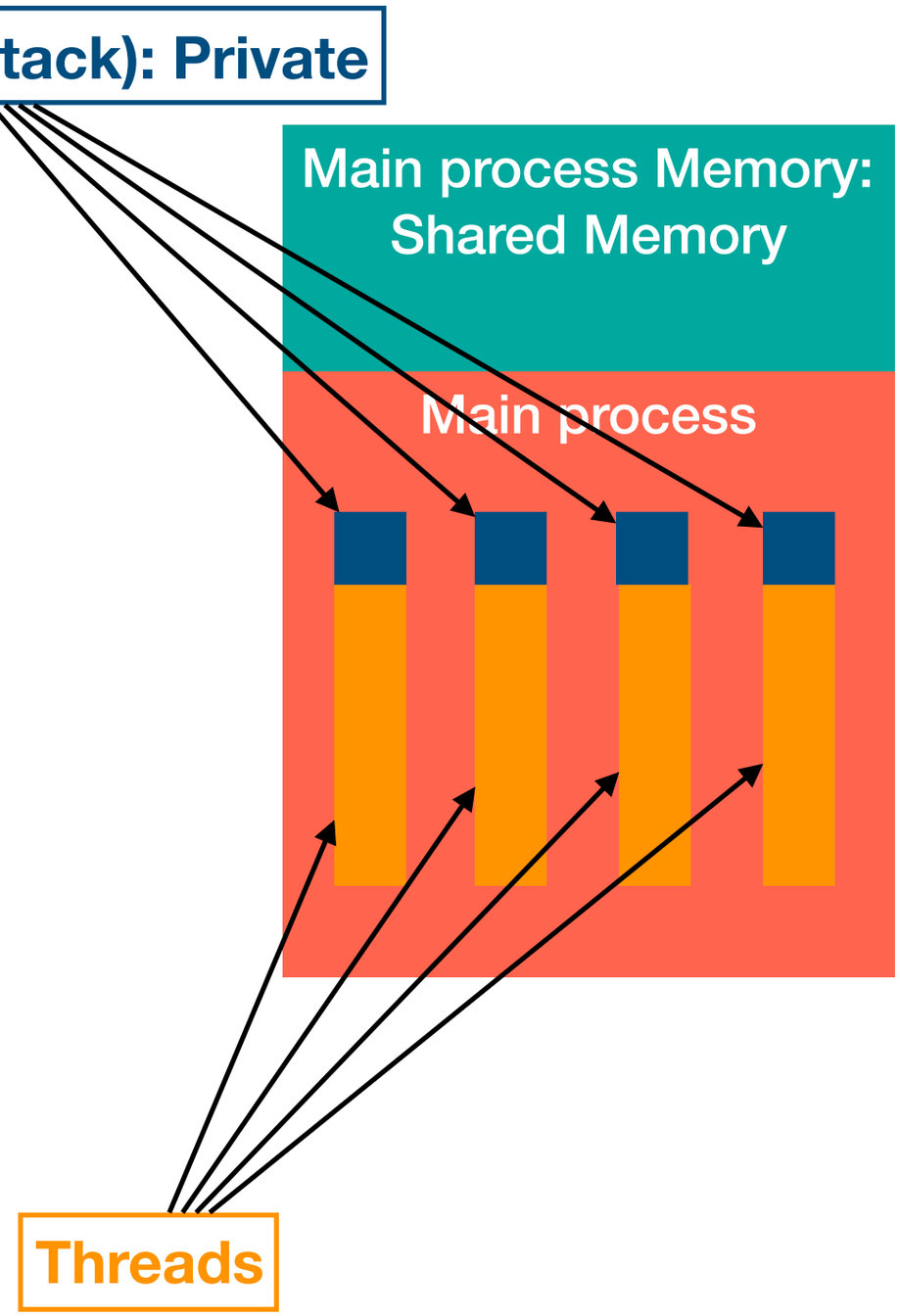
Concepts: Threads

- Each thread executes its own sequence of instruction corresponding to its implicit task
- The operating system **schedule** the execution of the threads on the available PU
- The overall order of execution is non-deterministic



Concept: Memory

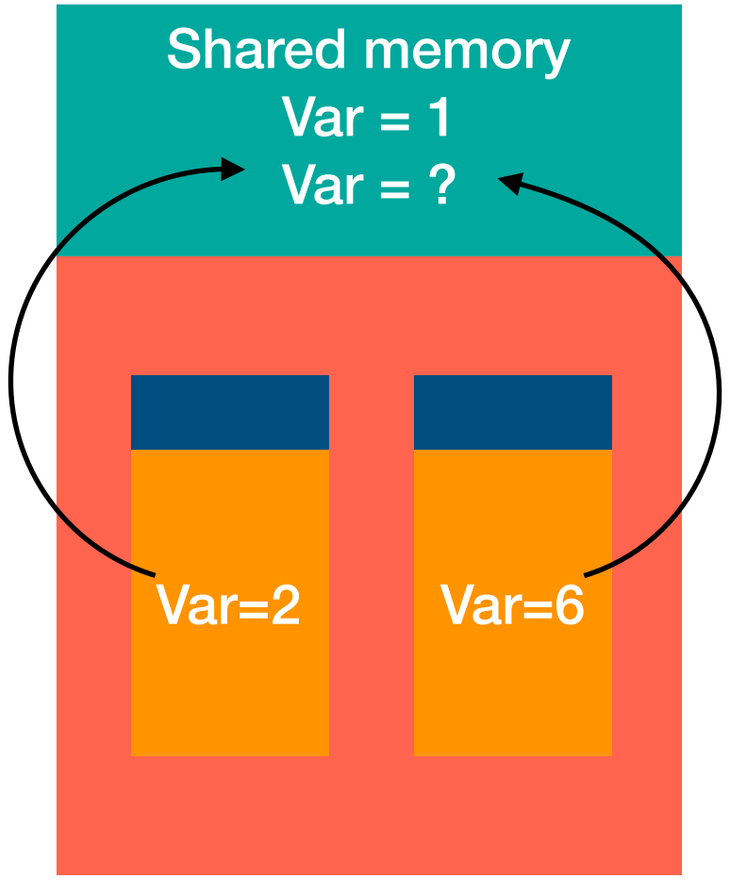
- Each task of the program share the memory space of the initial task (**shared memory**)
- Each tack dispose of a local memory space: the **stack**
- **Shared variables** are stored in shared memory
- **Private variables** are stored in the stack



Concept: Synchronisation

- In shared memory we have to introduce **synchronisation** between concurrent tasks

- Synchronisation ensures that 2 threads do not modify the same shared variable

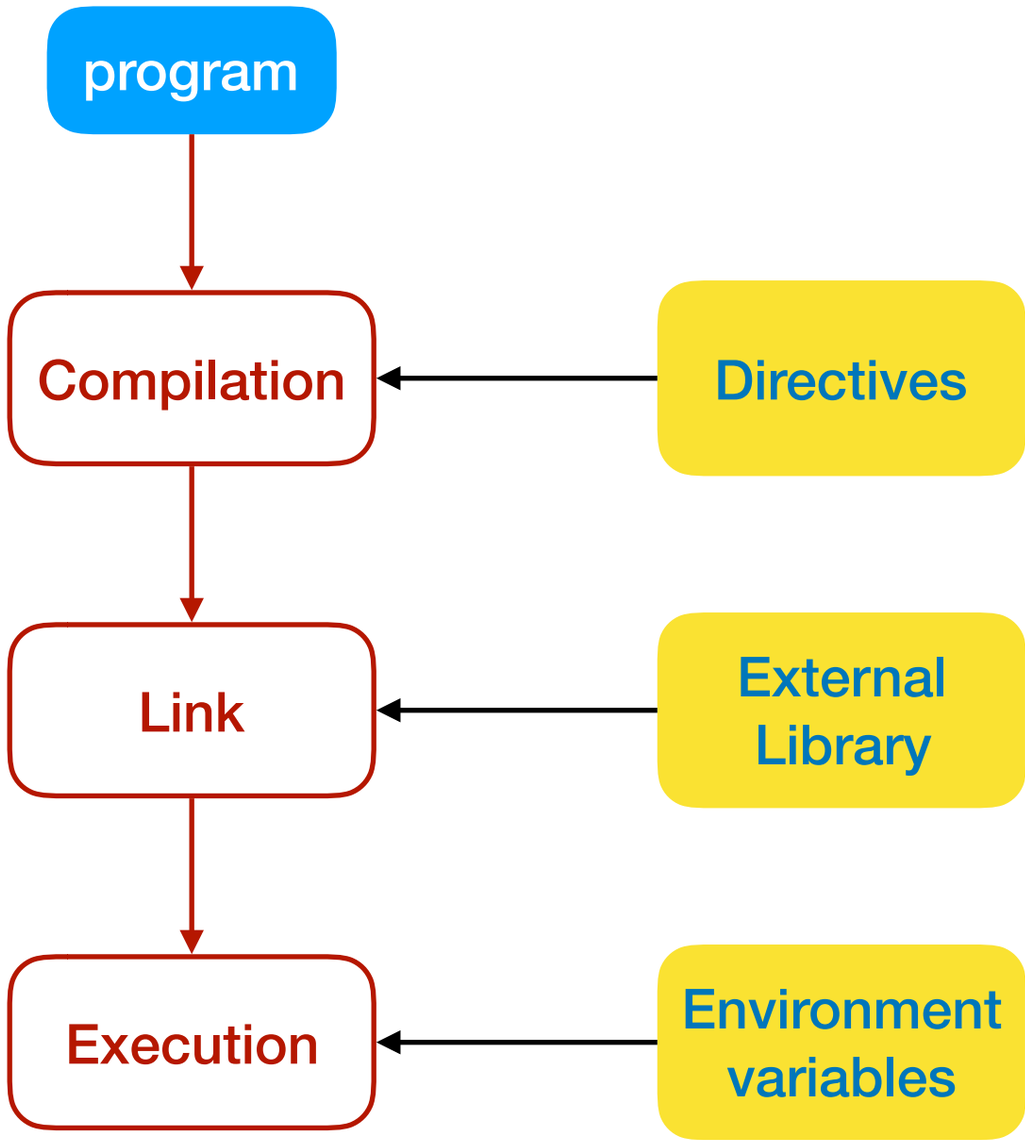


Concept: functionalities

- OpenMP permit parallel algorithms in shared memory by providing mechanisms to:
  - **Share work** between tasks
  - **Share or privatise data**
  - **synchronise threads**
  - **Assign explicit tasks**
  - **offload** part of the **work** to an accelerator

Concept: programming

- OpenMP available for C,C++ and Fortran
- OpenMP use ***directive*** and ***clause***
- Behavior of parallelism is set using environment variables



Directives

- ***Directives*** are define in the source code
- It is a comment line which be interpreted by the compiler if the openmp option is specified
- An openMP directive has the following general form:

```
sentinelle directive [clause …]
```

- The sentinel is a character string depending on the language used
- OpenMP provide some API functions, the header file omp.h (or the module OMP_LIB in Fortran) define the prototype of the OpenMP functions
  - two main api functions:
    - omp_get_num_threads() : get the number of threads in the parallel region
    - omp_get_thread_num(): get the id number of a thread in the parallel region
- Compilation
- Execution
  - Use environment variable to define the number of threads
  - Execute the code like any other

```
#include <omp.h>
. . .
#pragma omp parallel private(a,b)
{. . .}
```

```
!$ use OMP_LIB
. . .
!$OMP PARALLEL PRIVATE(a,b)
. . .
!$OMP END PARALLEL
```

```
gcc -fopenmp prog.c
export OMP_NUM_THREADS = 4
./a.out
```

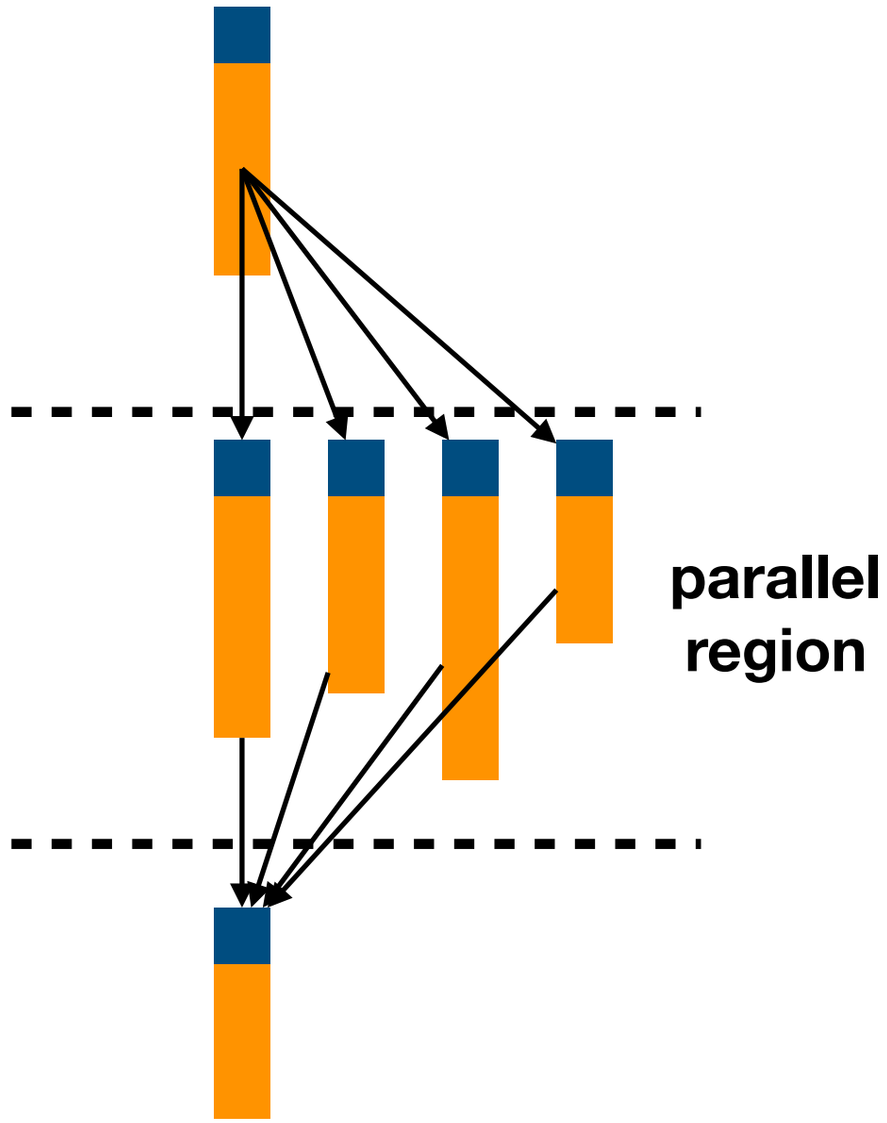
Parallel Region

- OpenMP is based on the fork-join model: alternate sequential and parallel region

- At the entry of a parallel region the master thread creates/ activates threads and implicit task for each

- During the parallel region each thread executes its implicit task

- At the end of the parallel region (joins) each thread disappears or hibernate



Parallel Region

- In the code a parallel region is defined using the directive **`parallel`**

- Whitin the same parallel region each task execute a separate implicit task but the tasks are composed of the same duplicate code

- The data-sharing attribute (DSA) of the variables are shared by default

- There is an implicit synchronisation barrier at the end of a parallel region

```c
#include <omp.h>
int main (int argc, char *argv[])
{
  int a, p;
  a=9000;
  p=0;
  /* Fork a team of threads */
  #pragma omp parallel
  {
  p=omp_in_parallel();
  printf("a = %d \n", a);
  }  /* end of parallel region,
        all threads join master */
  printf("p = %d \n", p);
}
```

```
gcc -fopenmp omp_first.c
export OMP_NUM_THREADS=4
./a.out
```

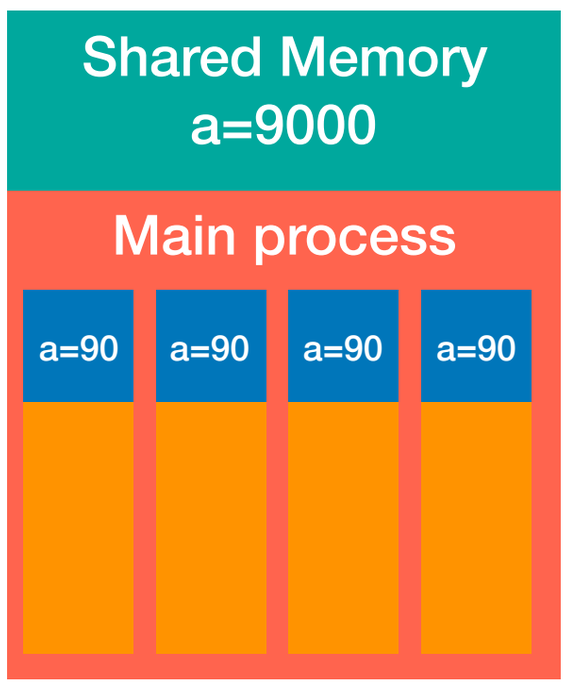
```
a = 9000
a = 9000
a = 9000
a = 9000
p = 1
```

Data-sharing attribute

- Private variables
  - **`private`** clause allows changing the DSA of a variable to private
  - If a variable has a private DSA it is allocated int the stack of each task
- The private variables are not initialized on entry the parallel region
- the **`firstprivate`** clause force the initialization of a private variable to the last value before entry parallel region
- the default clause allow to change the implicit DSA of variables
- In C and C++, variables declared inside a parallel region are private

```c
int main (int argc, char *argv[])
{
  int a, tid;
  a=9000;
  /* Fork a team of threads */
  #pragma omp parallel private(a,tid)
  {
  tid=omp_get_thread_num();
  a = a + 90 ;
  printf("thread id = %d, a = %d \n", tid,a);
  }  /* end of parallel region,
       all threads join master */
  printf("outside region a = %d \n", a);
}
```

```
thread id = 2, a = 90
thread id = 0, a = 90
thread id = 3, a = 90
thread id = 1, a = 90
outside region a = 9000
```



Work Sharing

- Parallel region allow to define portion of code than executed in parallel
- How to distribute/share work ?
- Two main cases:
  - independent loop
  - independent code sections

Parallel loop

- A loop is parallelizable if all iterations are independent
- parallelization by distribution of iterations on each thread
- the parallelization is defined by the directive **`for`**
  - infinite loop (while-like) are not parallelizable with this directive
- The distribution mode (how to the iterations are distributed on each thread) can be specified with the **`schedule`** clause
- control variables (loop indices) are private by default

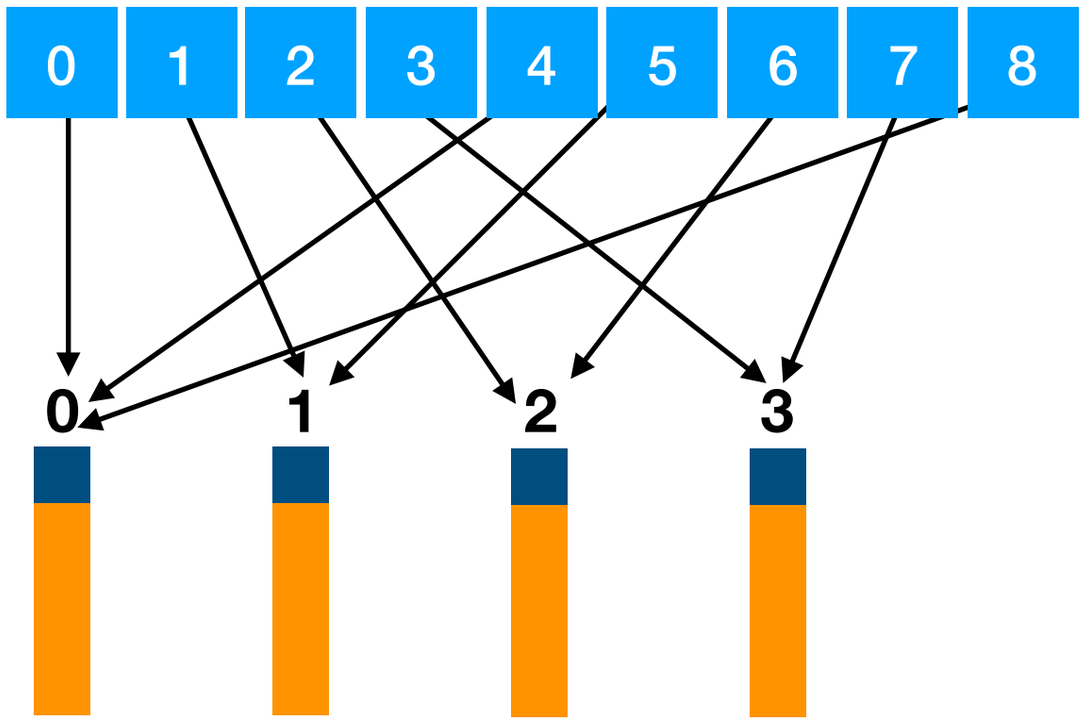
```
#pragma omp parallel
#pragma omp for private(i) schedule(static)
for (i=0;i<m;i++)
{
    C[i][j] =C[i][j] + A[i][k]*B[k][j];
}
```

Parallel loop: schedule mode

Static

- distribution consists of splitting the set of iterations of a set of chunks of given size

- The chunks are assigned to the threads in a cyclical manner (round-robin) following the order of the thread until all chunks have been distributed

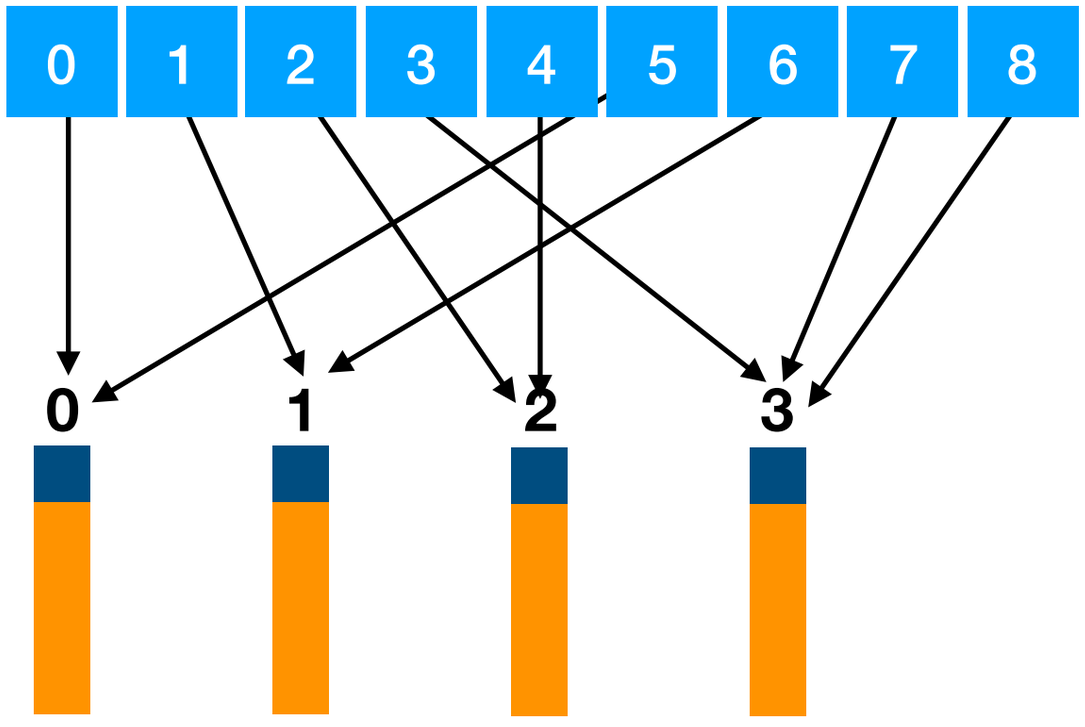


Parallel loop: schedule mode

Dynamic

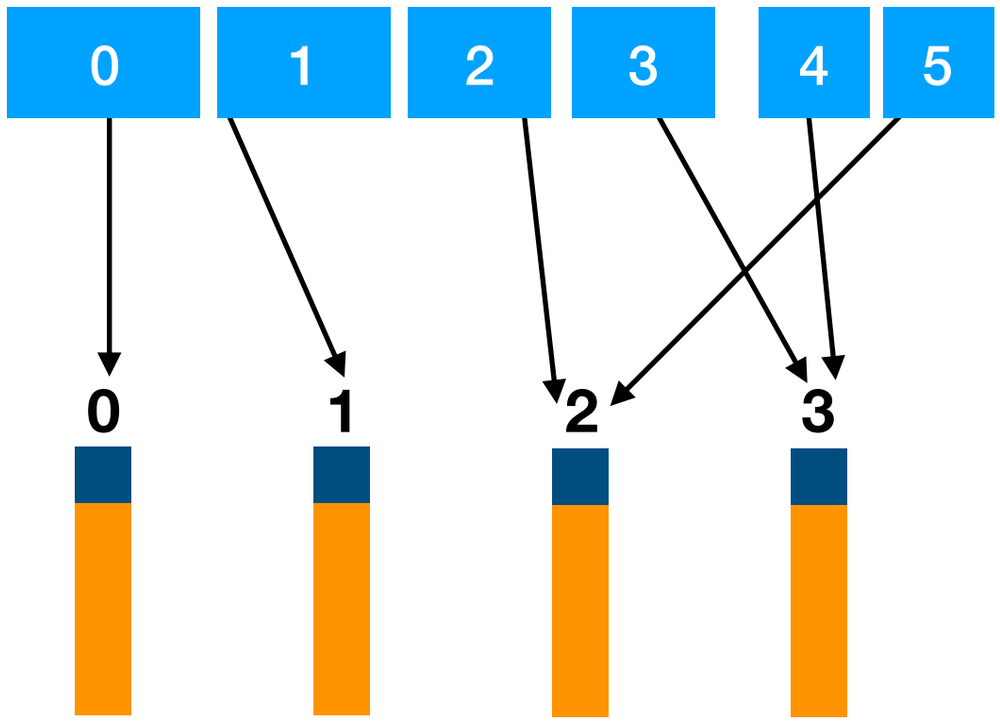
- distribution consists of splitting the set of iterations of a set of chunks of given size

- As soon as a thread is available (i.e. do not process any chunk) a new chunk is assign to it until all chunks have been distributed



Parallel loop: schedule mode

- Guided
  - distribution consists of splitting the set of iterations of a set of chunks of exponentially decreasing sizes
- The chunks are distributed using the dynamic mode



Parallel loop: schedule mode

Auto

- distribution mode choice is delegated to the compiler or to the execution system (runtime)

```
#pragma omp for private(i) schedule(runtime)
for (i=0;i<m;i++) {
    C[i][j] =C[i][j] + A[i][k]*B[k][j];
}
```

Runtime

- the choice of the schedule mode is deferred until the execution by using the OMP_SCHEDULE variable environment

```
export OMP_NUM_THREADS=4
./a.out
Temps de restitution :          141.5594 sec.
export OMP_SCHEDULE=« static"
./a.out
Temps de restitution :            4.0968 sec.
```

Parallel Loop: Reduction

- Reduction: associative operation applied to a shared variable

- clause **`reduction`**

- Kind of operation
  - Arithmetic:+,×
  - boolean

- Each thread calculates a partial result, the reduction is perform at the synchronization

```
int main (int argc, char *argv[])
{
int   i, n;
float a[100], b[100], sum;

/* Some initializations */
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i < n; i++)
     sum = sum + (a[i] * b[i]);

printf("   Sum = %f\n »,sum);
}
```

```
export OMP_NUM_THREADS=1
./a.out
   Sum = 328350.000000
export OMP_NUM_THREADS=4
./a.out
   Sum = 40843.000000
```

Parallel Sections

- A section is a portion of code executed by only one thread
- Several section can be define using the **`section`** directive
- All the sections are describe inside a block define using the **`sections`** directive
- The goal is to distribute several independent sections to different threads: each section define explicitly a task for its associated thread

```
#pragma omp parallel private(i)
  {
  #pragma omp sections
     {
     #pragma omp section
        {
        for (i=0; i<N; i++)
        {
          c[i] = a[i] + b[i];
        }
        }
    #pragma omp section
        {
        for (i=0; i<N; i++)
        {
          d[i] = a[i] * b[i];
        }
        }
    }  /* end of sections */

  }  /* end of parallel section */
```

Exclusive execution

- Some portion of code included in a parallel region need to be executed by only one thread
- Directive **`single`** and **`master`**

Single

- The single directive allows executing portion code by only one thread
- The thread executing the portion is non-deterministic
- The end of a single region is an implicit barrier for each threads

Master

- The master directive allows executing portion code only by the master thread
- Begin and end of a master region are **NOT** a barrier

```
#pragma omp parallel private(a,tid)
 {
 tid=omp_get_thread_num();
 a = a + 90 ;
 #pragma omp single
 {
   a = 10+tid ;
 }
 printf("thread id = %d, a = %d \n", tid,a);
 }  /* end of parallel region */
```

```
thread id = 0, a = 90
thread id = 3, a = 90
thread id = 1, a = 11
thread id = 2, a = 90
```

```
thread id = 1, a = 90
thread id = 0, a = 90
thread id = 2, a = 12
thread id = 3, a = 90
```

```
#pragma omp parallel private(a,tid)
 {
 tid=omp_get_thread_num();
 a = a + 90 ;
 #pragma omp master
 {
   a = 10+tid ;
 }
 printf("thread id = %d, a = %d \n", tid,a);
 }  /* end of parallel region */
```

```
thread id = 0, a = 10
thread id = 1, a = 90
thread id = 3, a = 90
thread id = 2, a = 90
```

```
thread id = 3, a = 90
thread id = 0, a = 10
thread id = 2, a = 90
thread id = 1, a = 90
```

Synchronization

- Synchronization become necessary in the following case
- Ensure that all threads reach the same instruction point (barrier mechanism)
- Order execution of all concurrent threads to prevent race condition (mutual exclusion mechanism)
- Synchronize a part of the threads (lock mechanism)

Synchronization

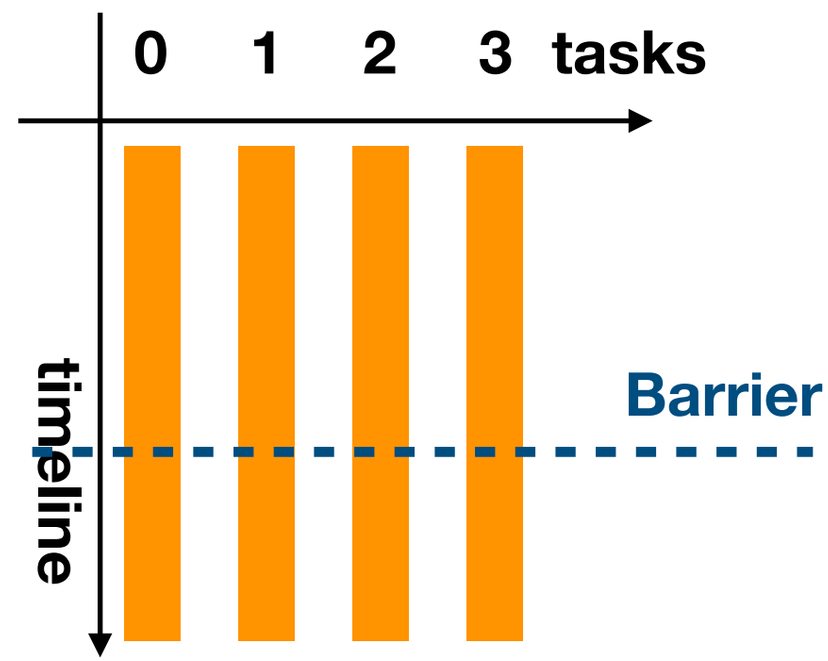
- By default each end of parallel loop is an implicit synchronization
- The clause **`nowait`** avoid this synchronization
- The **`barrier`** directive allow an explicit synchronization
- The mutual exclusion (one task at time) is implicitly applied in reduction

Synchronization: Barrier

- The **`barrier`** directive synchronize all threads in a parallel region

- Each thread waits until all reach the barrier then continues the task execution

```
#pragma omp parallel
{
#pragma omp master
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
#pragma omp barrier
#pragma omp for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
}
```



Synchronization: atomic

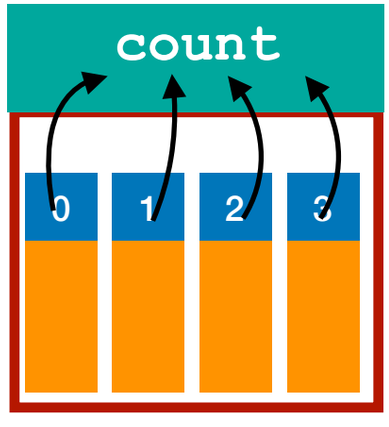
- The **`atomic`** directive ensures that a shared variable is read/write in memory by only one thread at time

- /!\ each thread execute the instruction (not single directive)

```
int main (int argc, char *argv[])
{
  int count, tid;
  count=0;
  #pragma omp parallel private(tid)
  {
  tid=omp_get_thread_num();
  #pragma omp atomic
  count = count + 1 ;
  printf("thread id = %d, count = %d \n", tid,count);
  }
  printf("outside region count = %d \n", count);
}
```

```
thread id = 0, count = 1
thread id = 1, count = 2
thread id = 3, count = 3
thread id = 2, count = 4
outside region count = 4
```

```
thread id = 0, count = 1
thread id = 3, count = 4
thread id = 1, count = 2
thread id = 2, count = 3
outside region count = 4
```



Synchronization: critical

- critical region is generalization of atomic directive
- directive **`critical`**
- All the threads execute the region in non-deterministic order only one at time
- A critical region can be explicitly named
- several critical region with the same name are considered as only one critical region

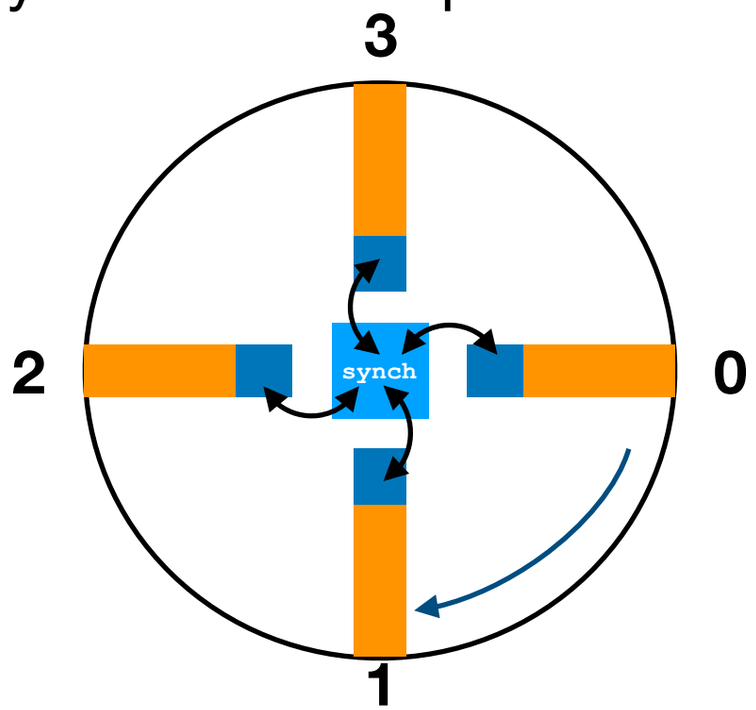
```
#pragma omp parallel private(i, j, err)
{
  #pragma omp for schedule (static)

   for( i=1 ; i<=ni ; i++ ) {
     for( j=1 ; j<=nj ; j++ ) {
       if( Phi[i][j] > 1.e-6 )
         err=dmax(fabs(Phi[i][j]/Phio[i][j]-1.0),err );
       else
         err=dmax(fabs(Phi[i][j]-Phio[i][j]    ),err );
     }
   }

   #pragma omp critical
   errmax = dmax( errmax, err );
}
```

Synchronization: flush

- flush directive refresh the value of a shared variable in global memory

- Useful because many machine has hierarchical memory with multiple level of cache

- Can also serve to establish a synchronization point




```
int main (int argc, char *argv[])
{
  int i,synch, nthreads, tid;
  synch=0;
  /* Fork a team of threads */
  #pragma omp parallel private(tid,nthreads)
  {
  tid=omp_get_thread_num();
  nthreads=omp_get_num_threads();
  if (tid == 0)
  {
     for (i=0;i<10000;i++)
     {
        #pragma omp flush(synch)
        if (synch == nthreads-1) break;
     }
     printf("master = %d, i = %d \n", tid,i);
  }
  else
  {
     for (i=0;i<10000;i++)
     {
        #pragma omp flush(synch)
        if (synch == tid-1) break;
     }
     printf("thread id = %d, i = %d \n", tid,i);
  }
  printf("thread id = %d, synch = %d \n", tid,synch);
  synch=tid;
  }
}
```

Task Parallelism

- The fork-join (thread centered) model is limited
- In particular absolutely not adapted to dynamic problem (while-loop, tree research, …) or recursive algorithms
- OpenMP 3.0 and 4.0 provide parallelism task centered
- Allow parallelism for recursive or pointer-based algorithm, commonly used in C and C++

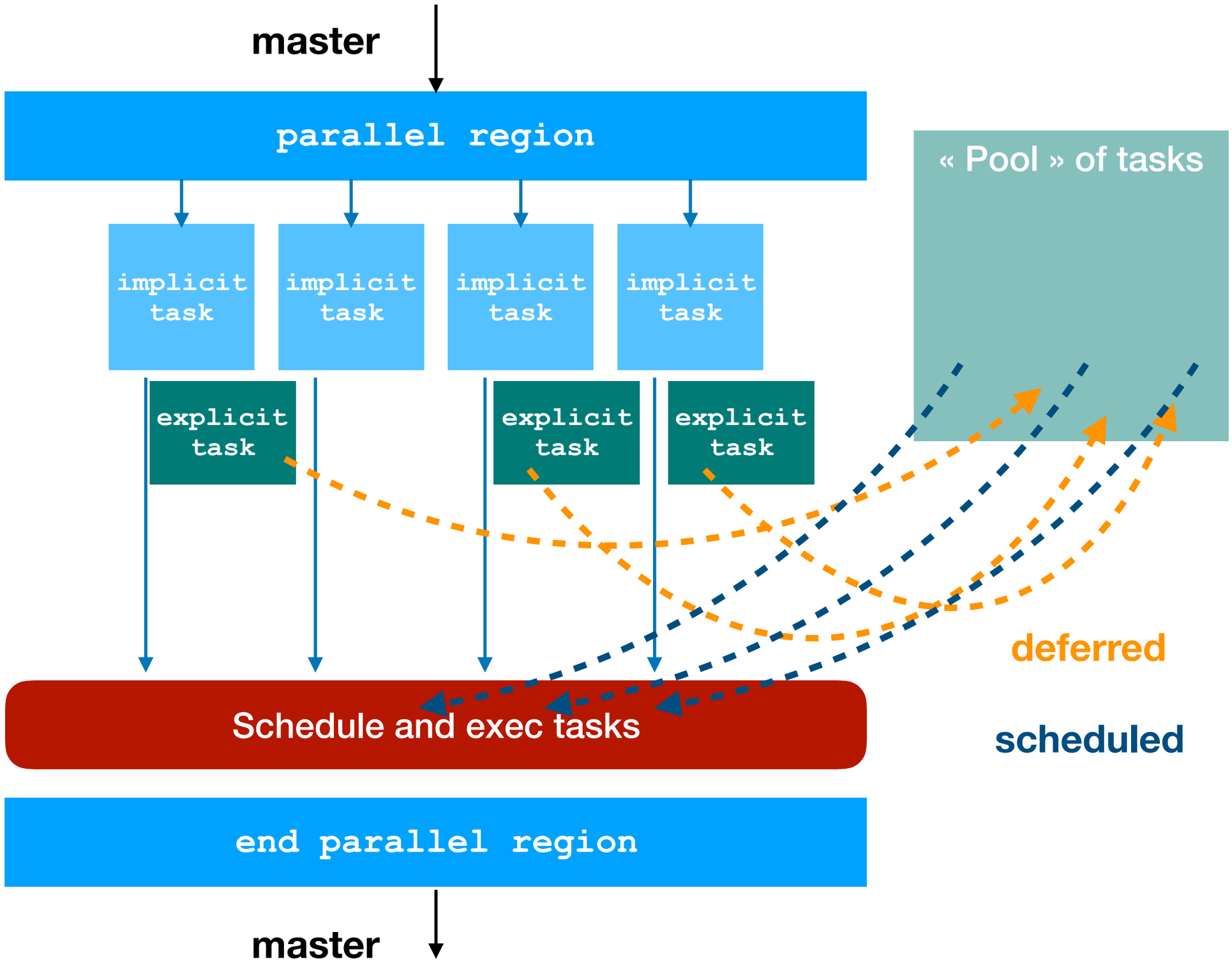
Tasks: Concept

- As already view: a task is an instance of executable code and its associated data executed by one thread
- Two type of task
  - implicit task generated by the parallel directive
  - explicit task generated by the task directive

Task: execution model

- Execution model begin with the master thread
- In parallel region (parallel):
    - creation of a team of threads
    - creation of implicit tasks, one per thread, each thread executing its implicit task
- In task region
    - Creation of explicit tasks
    - the execution of this task can be deferred
- Execution of explicit tasks
    - At the task scheduling point (task, taskwait, barrier) the available thread begin executing waiting tasks
    - a thread can switch from the execution of one task to another one
- At the end of parallel region:
    - all the task finish their execution

Task: execution model



Tasks examples

- Each thread create a task with the printf

```
#pragma omp parallel
{
#pragma omp task
    printf(« greeting from a random thread\n");
}
```

- Only one thread create the two task

```
#pragma omp parallel
{
#pragma omp single
{
#pragma omp task
    printf("hello world\n");

#pragma omp task
    printf("hello again!\n");
}
}
```

Dependencies

- **`depend(type_dependance:list)`** clause allows managing dependencies between tasks having the same parent

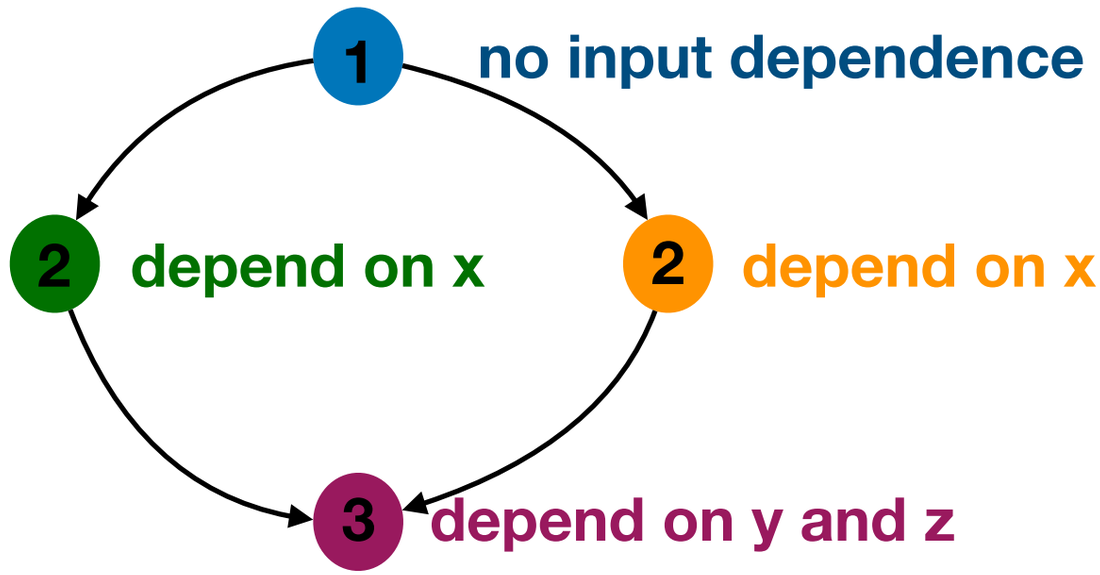
- A task T1 depending on T2 cannot been executed before T2 has terminated

- Dependencies are defined with **`in`** and **`out`**

  - **`in`** « variables » expected in input

  - **`out`** « variables » returned as output

```
#pragma omp parallel
{
#pragma omp single
{
     int x, y, z;
#pragma omp task depend( in: x ) depend( out: y)
     y = f(x);
#pragma omp task depend( out: x )
     x = init();
#pragma omp task depend( in: x ) depend( out: z)
     z = g(x);
#pragma omp task depend( in: y, z )
     finalize(y, z);
}
}
```



Exercises

parallel loop

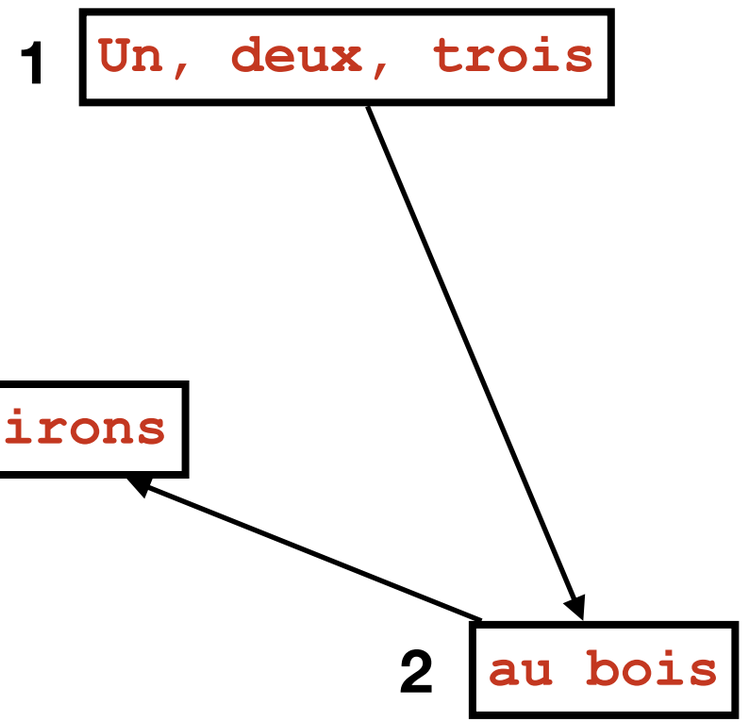
- place the omp directive with the good clause

```
int main (int argc, char *argv[])
{
int    i, n, ilast;
int a[100000];

/* Some initializations */
n = 100000;
for (i=0; i < n; i++)
  a[i] = i ;

#pragma omp parallel for private(ilast)
  for (i=0; i < n; i++){
    if ((a[i]%10000)== 0){
       ilast=i;
       printf("value found at ilast = %d \n",ilast);
    }
  }
}
```

Task Scheduling

- Give the task schedule for the following code

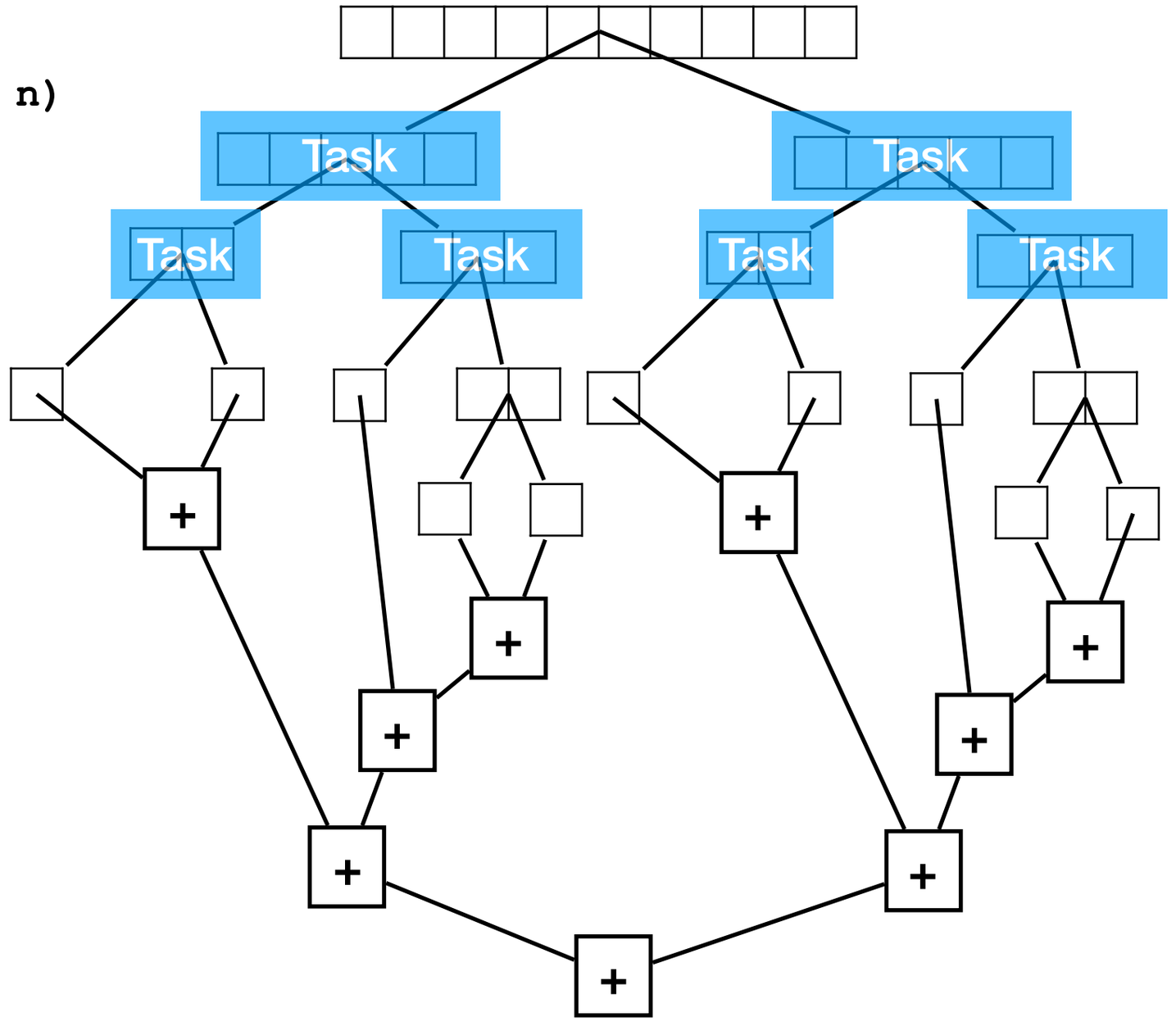
```
int main (int argc, char *argv[])
{
  #pragma omp parallel
  {
  #pragma omp single
  {
  #pragma omp tasks depend(out:t1)
  printf("Un, deux, trois \n");
  #pragma omp tasks depend(in:t2)
  printf("Nous irons\n");
  #pragma omp tasks depend(in:t1,out:t2)
  printf("au bois\n");
  }
  }
}
```



Divide and conquer

- How to parallelize this code (with openmp) ?

```
float sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return *a;
    }
    // recursive case
    size_t half = n / 2;
    float x,y;
    x=sum(a,half);
    y=sum(a+half,n-half);
    x+=y;
    return x;
}
```



Divide and conquer

```
float omp_sum(const float *a, size_t n)
{
    // base cases
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return *a;
    }
    // recursive case
    size_t half = n / 2;
    float x, y;
    #pragma omp parallel
    {
      #pragma omp single nowait
      {
        #pragma omp task shared(x)
        x=sum(a,half);
        #pragma omp task shared(y)
        y=sum(a+half,n-half);
        #pragma omp taskwait
        x+=y;
      }
    }
    return x;
}
```

Vectorization

SIMD and Vectorization

- Multiple processing elements perform the same operation on multiple data points simultaneously

```
a = [1,9,2,8]
b = [3,4,5,6]
for i in range(4):
  Result[i] = a[i] * b[i]
```

a=	1	9	2	8
b=	3	4	5	6
Result=	3	36	10	48
iteration	1	2	3	4

SISD

```
a = [1,9,2,8]
b = [3,4,5,6]
Result[1:4] = a[1:4] * b[1:4]
```

a=	1	9	2	8
b=	3	4	5	6
Result=	3	36	10	48
iteration	1			

SIMD

Vector computer

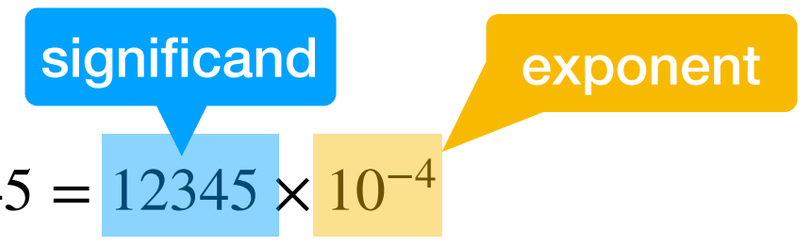
Vector computer: not exactly SIMD

Operation Segmentation

- Each scalar operation can be decomposed into atomic segment

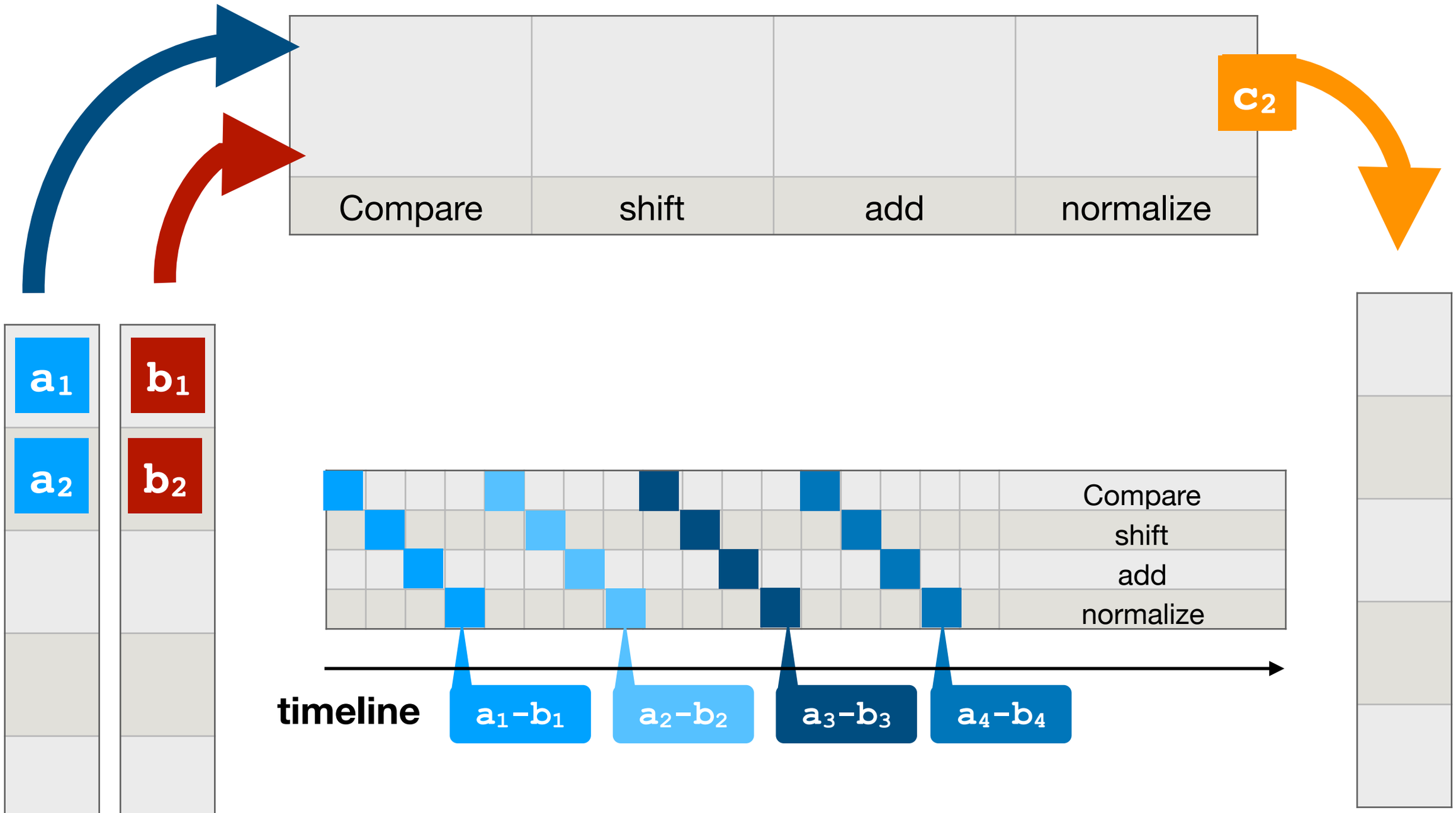
Example: floating point addition:

- floating point representation $1.2345 = 12345 \times 10^{-4}$

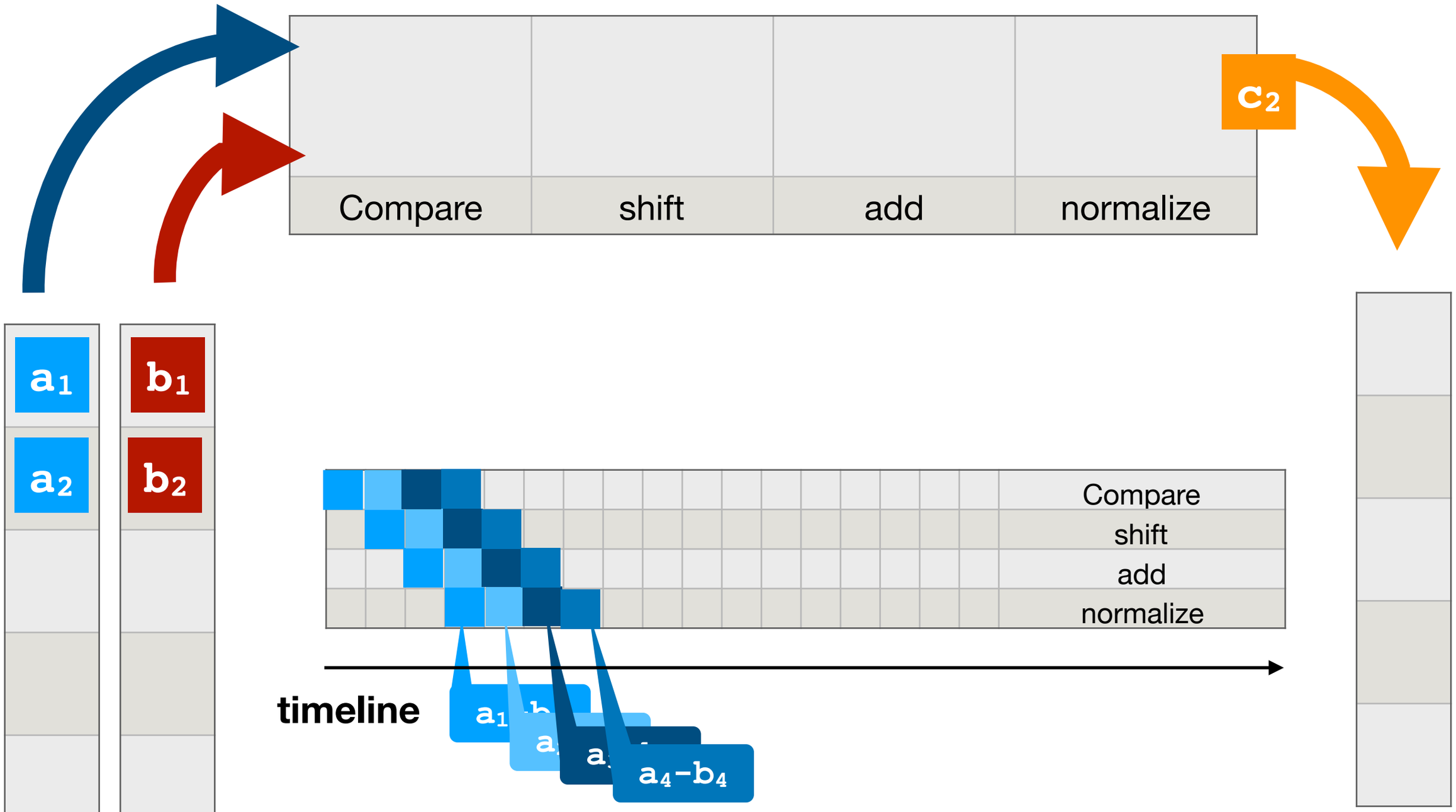


1. compare exponent `1.14e9 - 2.78e8`
2. shift significand (mantissa) `1.14e9 - 0.278e9`
3. add significand `0.862e9`
4. select exponent and normalize `8.62e8`

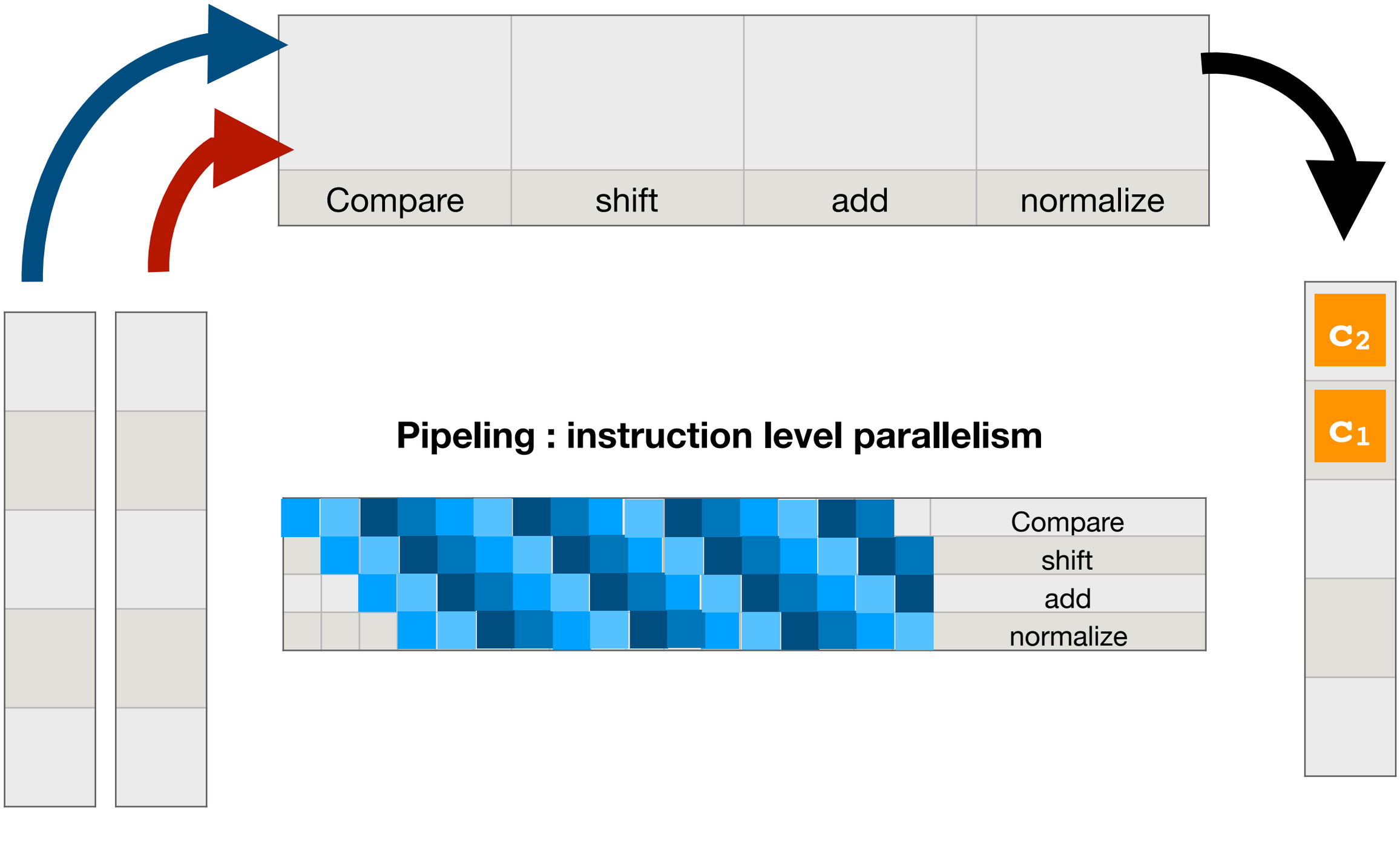
Vector computer: not exactly SIMD



Vector computer: not exactly SIMD



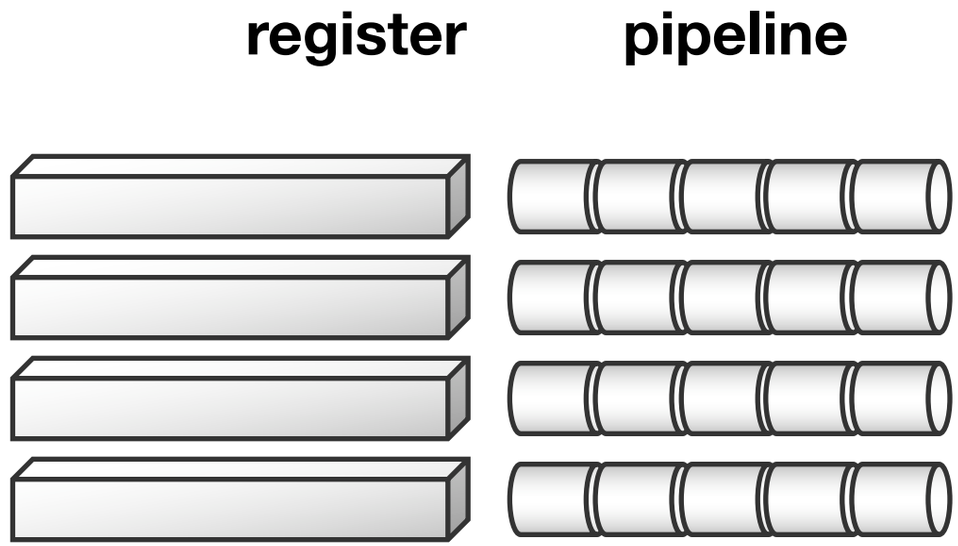
Vector computer: not exactly SIMD



Pipeling : instruction level parallelism

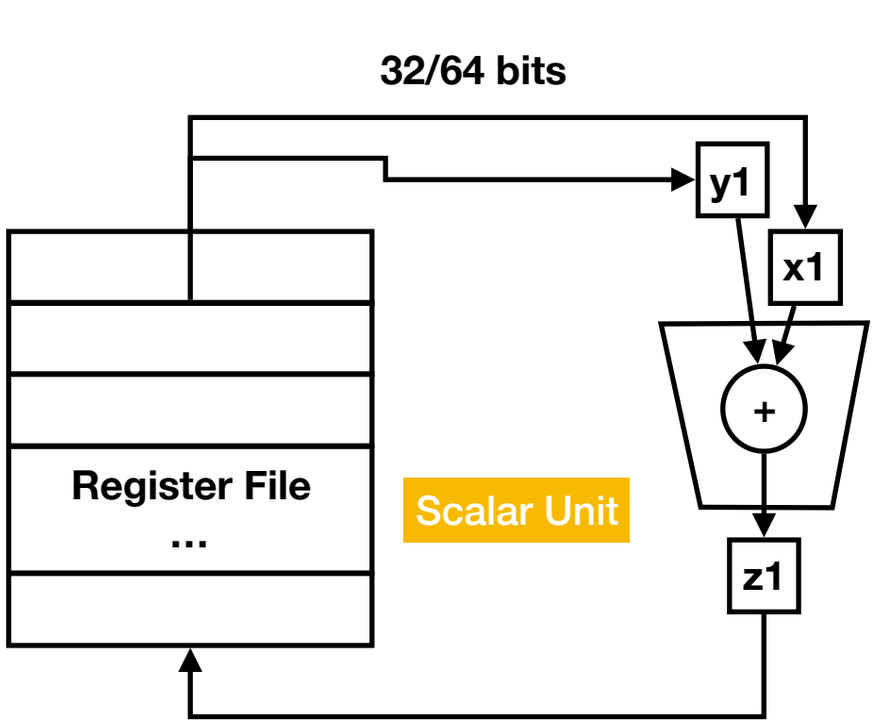
Vector computer

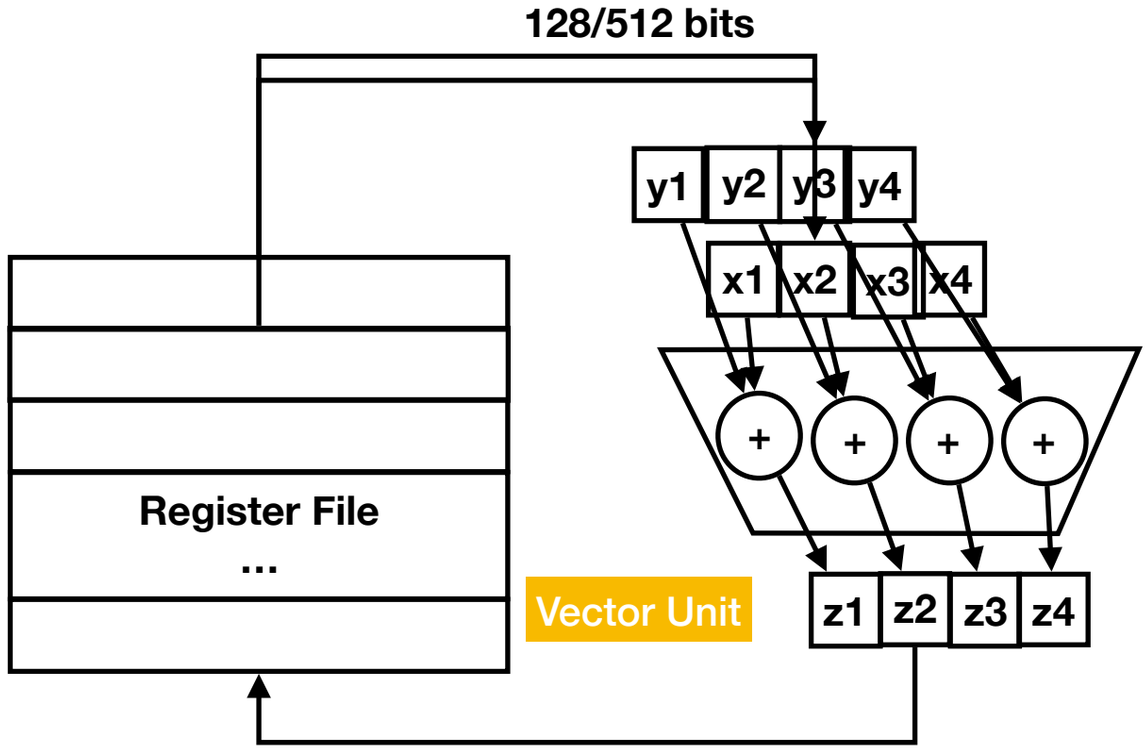
A lot of pipelines with big registers



Vector in standard architecture (intel-like)

Since the end of 90's





Evolution of vector hardware

Floating-Point

64-bit — **Double Precision**

32-bit — **Single Precision**

Tech. SSE/AVX 128

64-bit		64-bit	
32-bit	32-bit	32-bit	32-bit

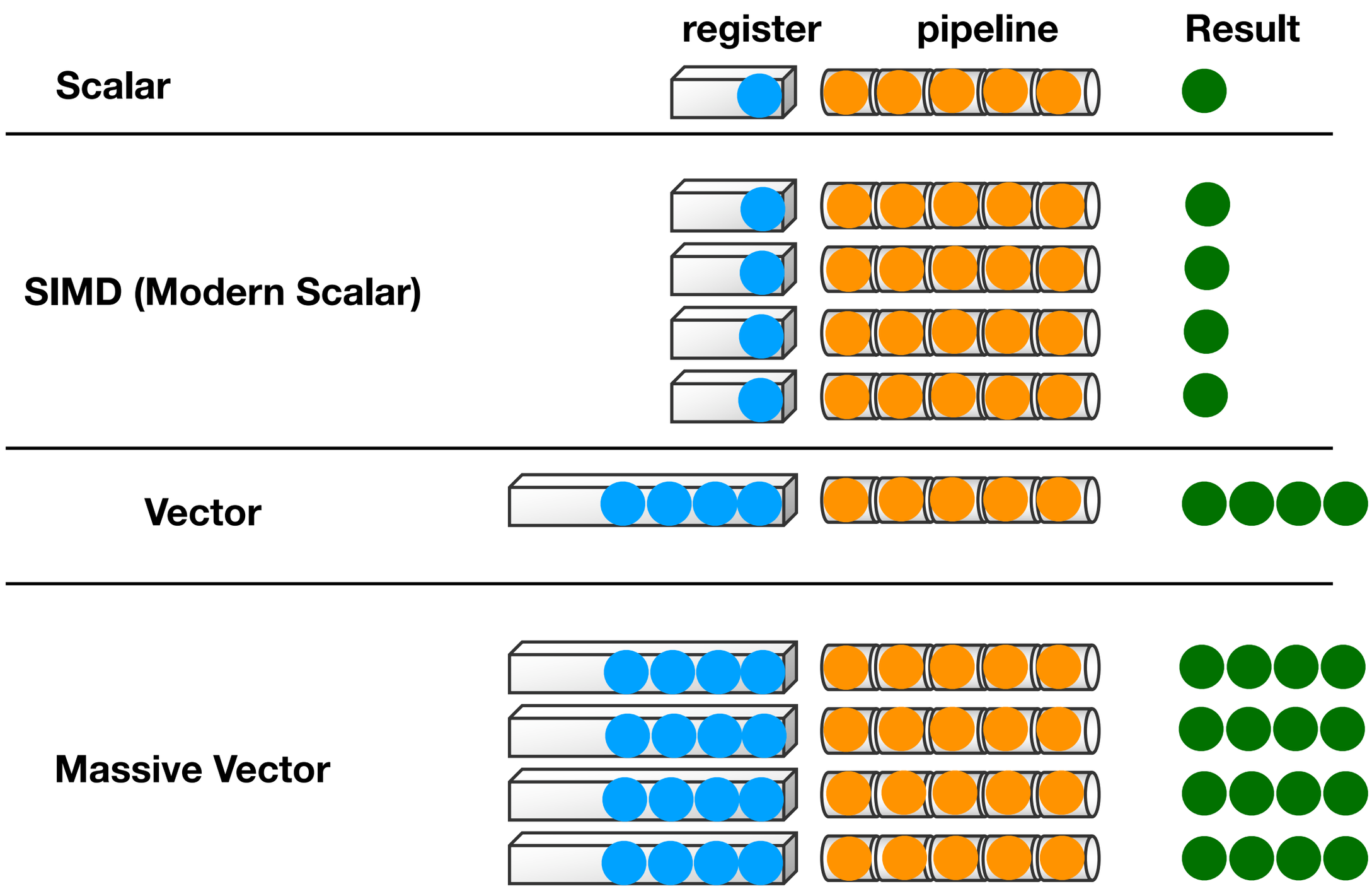
Tech. AVX 256

64-bit		64-bit		64-bit		64-bit	
32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit

Tech. AVX 512

64-bit		64-bit		64-bit		64-bit		64-bit		64-bit		64-bit		64-bit	
32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit

Resume


register
pipeline
Result
Scalar
SIMD (Modern Scalar)
Vector
Massive Vector

Vectorization programmation

How to vectorize ?

- 3 ways

    1. Very implicit way: you don’t know but compiler try to vectorize

    2. semi-explicit way: use directive to guide your compiler (openmp allow this kind of vectorization)

    3. explicit way: you explicitly use the data structure to develop vectorized source code

How to vectorize ?

- Implicit way: the compiler vectorize for you

```
icc -qopt-report prodmat.c
```

generate a report

```
LOOP BEGIN at prodmat.c(75,2)
   remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 3 2 1 )
   remark #15542: loop was not vectorized: inner loop was already vectorized   [ prodmat.c(75,2) ]

   LOOP BEGIN at prodmat.c(74,7)
      remark #15542: loop was not vectorized: inner loop was already vectorized

      LOOP BEGIN at prodmat.c(73,5)
         remark #15301: PERMUTED LOOP WAS VECTORIZED
      LOOP END
   LOOP END
LOOP END
```

```
./a.out
      Temps de restitution :           1.9161 sec.
      Resulat partiel : 1 2 ... 1799 1800
```

```
icc -no-vec prodmat.c
```

avoid vectorization

```
./a.out
      Temps de restitution :           3.3142 sec.
      Resulat partiel : 1 2 ... 1799 1800
```

How to vectorize ?

- Directive way (with openmp for example)

```
for ( int i = 0 ; i < m ; i++)
    #pragma omp simd
    for ( int j = 0 ; j < n ; j++)
        c[i] += A[i*n_pad+j] * b[j];
```

- Explicit way: use directly vector data structures and vector fonction in your code

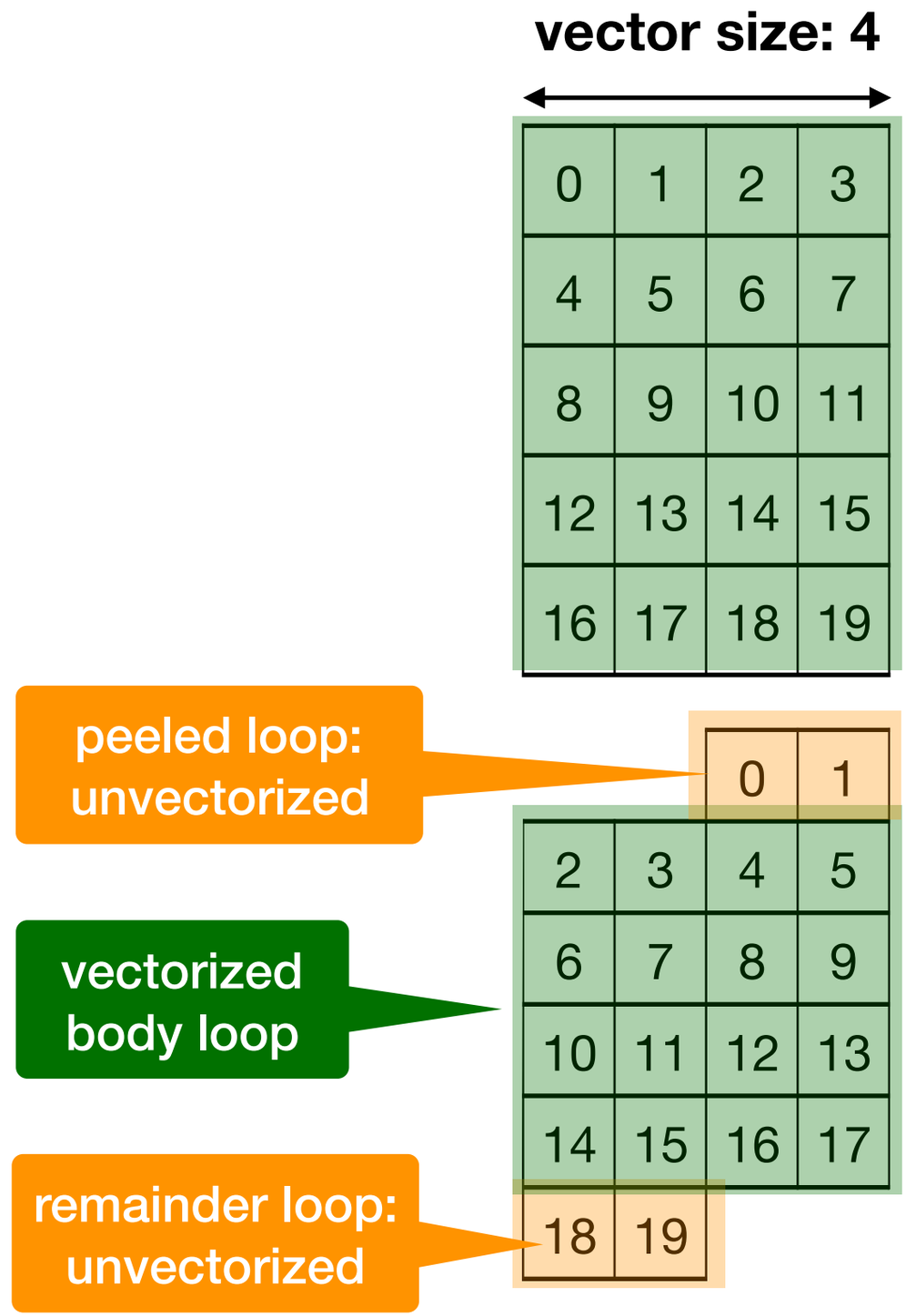
```
#include <emmintrin.h>
void vsum(int n,short *a,short *b,short *c)
{
    int i;
    __m128i *va=(void*)a,*vb=(void*)b;
    __m128i *vc=(void*)c;
    for(i=0;i<n/8;i++)vc[i]=_mm_add_epi16(va[i],vb[i]);
}
```

- A little bit unreadable
- unportable code: what will happen on another architecture ?
- not recommended by constructor

Vectorization issues

Data alignment

- Aligned load
  - Address is aligned
  - One cache line
  - One instruction
  - 1 or 2 kernel version : vector/remainder
- Unaligned load
  - Address is not aligned
  - Potentially multiple cache lines
  - Potentially multiple instructions
  - 3 kernel version peel/vector/remainder



Vectorization issues

Data dependencies

```
for (i=0;i<N;i++) = 1, N
  A(i + m) = A(i) + B(i);
```

Indirection

```
for (i=0; i<N; i++)
  A[B[i]] = C[i]*D[i];
```

function call

```
for (i = 1; i < nx; i++) {
  x = x0 + i * h;
  sumx = sumx + func(x, y, xp);
}
```

external/internal loop

```
for(i = 0; i <= MAX; i++) {
  for(j = 0; j <= MAX; j++) {
    D[j][i] += 1;
  }
}
```

Branch

```
for(i = 0; i <= MAX; i++) {
  if ( D[i] < N)
    do_this(D);
  else if (D[i] > M)
    do_that();
}
```

little loop

```
for(int i = 0; i <uknw_small_value; i++)
    a[i] = z*b[i];
```

Focus on Python

- Use the good data structure

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [6, 7, 8, 9, 10]
>>> [x * y for x, y in zip(a, b)]
[6, 14, 24, 36, 50]
```

basic list

```
>>> import numpy as np
>>> na = np.array([1, 2, 3, 4, 5])
>>> nb = np.array([6, 7, 8, 9, 10])
>>> a * b
array([ 6, 14, 24, 36, 50])
```

numpy array

```
>>> a = [random.randint(1, 100) for _ in range(1000000)]
>>> b = [random.randint(1, 100) for _ in range(1000000)]
>>> %timeit res = [x * y for x, y in zip(a, b)]
164 ms ± 17.6 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
>>> import numpy as np
>>> na = np.random.randint(1, 100, 1000000)
>>> nb = np.random.randint(1, 100, 1000000)
>>> %timeit na * nb
2.89 ms ± 348 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- And the good instruction

```
>>> a = [random.randint(1, 100) for _ in range(1000000)]
>>> b = [random.randint(1, 100) for _ in range(1000000)]
>>> %timeit res = [x * y for x, y in zip(na, nb)]
363 ms ± 21.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Focus on Python: Numba

- Numba
    - Package allow optimization for numerical code:
        - just-in-time, inference, function compilation
            - function compilation
                - Numba can compile python function => function run faster
            - inference
                - By default there no type in python => python need to verify each operation (result depend of implicit type) ; numba allows type specification => python run faster (less control)
            - just-in-time
                - Numba allow generating optimized code without compilation step (generation done at each first call of the function or when the data type changes)
- numerical oriented
    - Numba is numerical oriented, numpy data structure are recommended for performance

Focus on Python: Numba

```
def prod(tabx,taby):
    res=np.zeros(tabx.size)
    for i in range(tabx.size):
        res[i]=tabx[i]*taby[i]
    return res
na = np.random.randint(1,100,1000000)
nb = np.random.randint(1,100,1000000)
%timeit prod(na,nb)
609 ms ± 125 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
from numba import vectorize
@vectorize
def numprod(x,y):
    res=x*y
    return res
%timeit numprod(na,nb)
3.01 ms ± 296 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

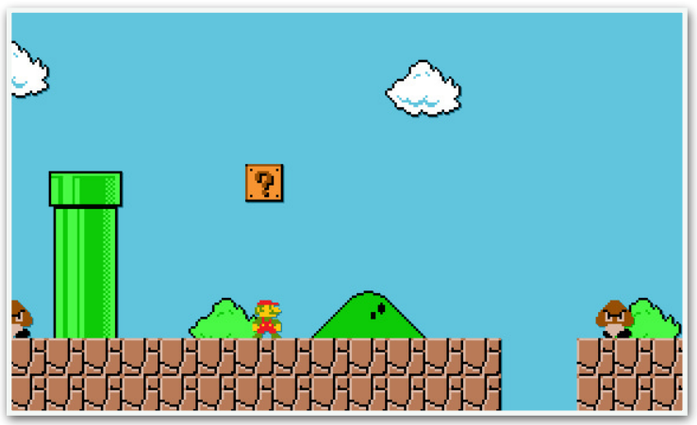
GPU Programming

GPU: Architecture Model

What is GPU ?

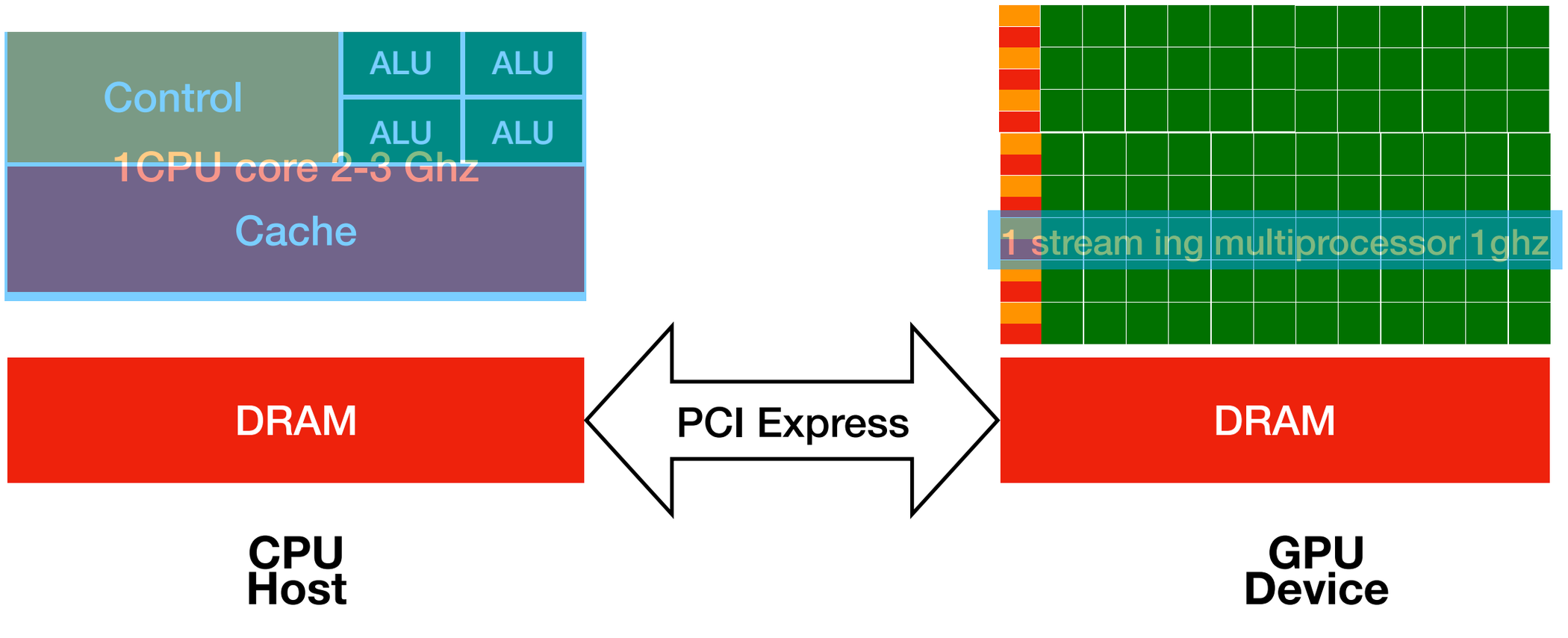
- Wikipedia : « graphics processing unit (**GPU**) is a **specialized electronic circuit** designed to rapidly manipulate and alter memory to **accelerate the creation of images** in a frame buffer intended for output to a display device »

- GPU is:
  - Accelerator for specialized problems
  - use SIMD paradigm (well design for image processing)
  - booming with video game



GPU for computing ?

- CPU vs GPU
- GPU comes from graphic rendering
    - Single Instruction Multiple Data (compute intensive & few control)
    - Throughput oriented (thousands of pixels simultaneously)
    - 1 SM comparable to 1 CPU core
    - Connected with PCI-Express (Host and Device)



Why GPU for computing ?

- Since 2000's frequency scaling is over… we are now scaling core

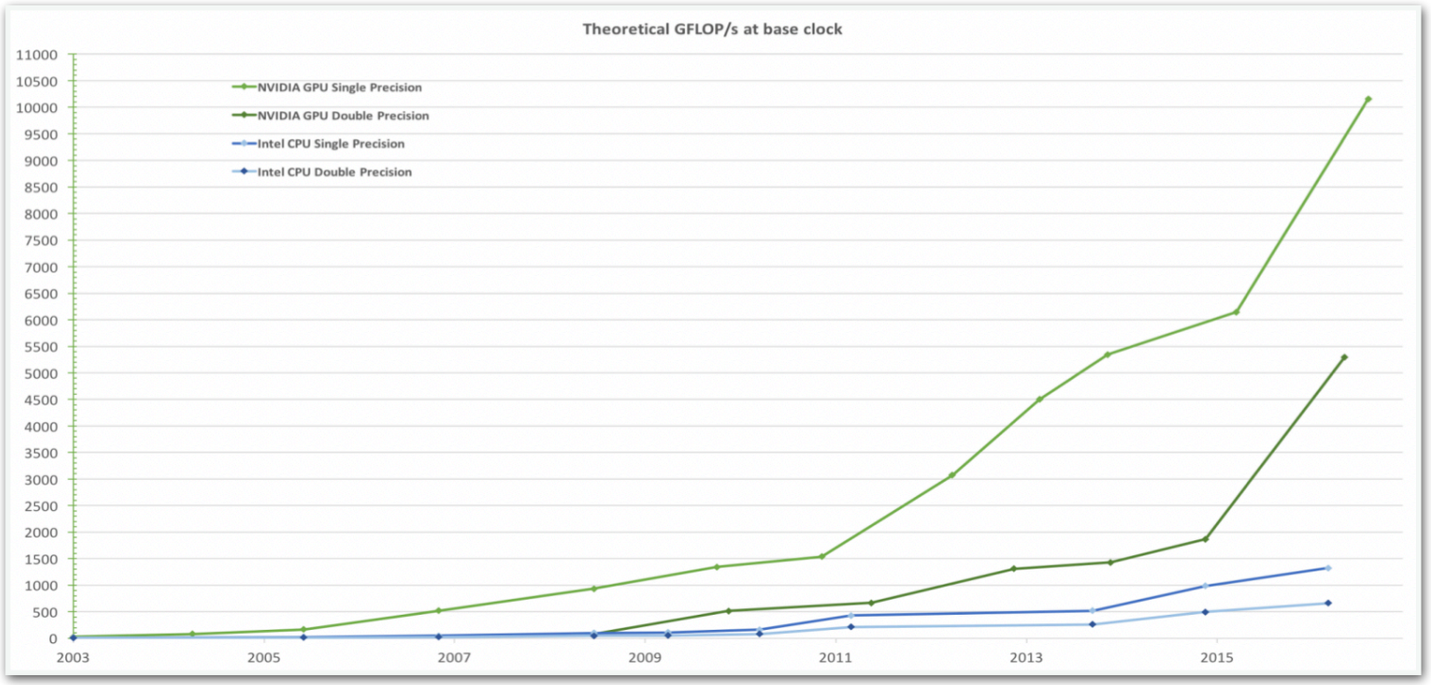
- GPU is a massively multi-threaded many-cores architecture

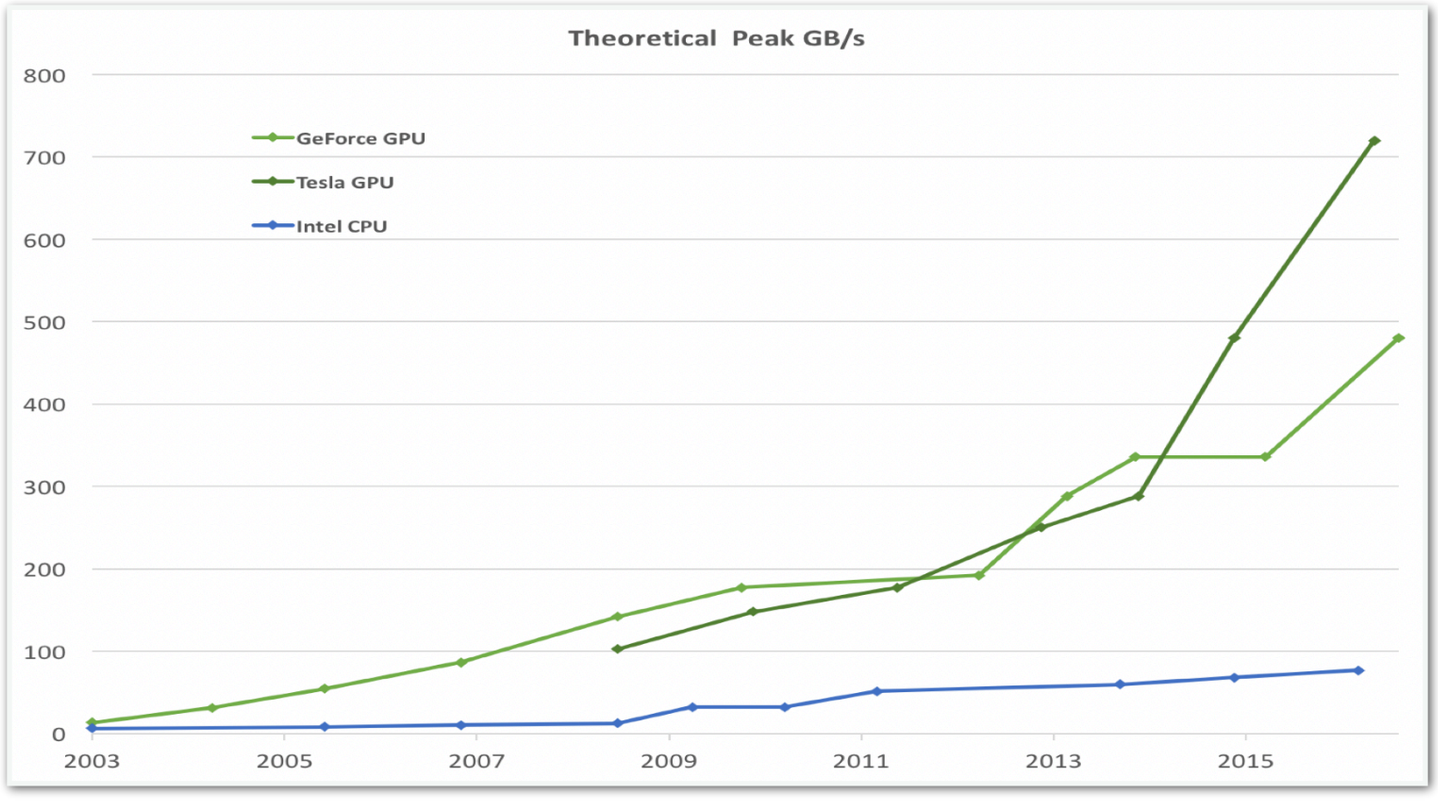
- Thousand of threads executable in parallel

Kepler 80 (K80)	2048*13=26 624 threads
Pascal 100 (P100)	2048*56=114 688 threads
Volta 100 (V100)	2048*80=163 840 threads

- Big market (low production price)

GPU faster

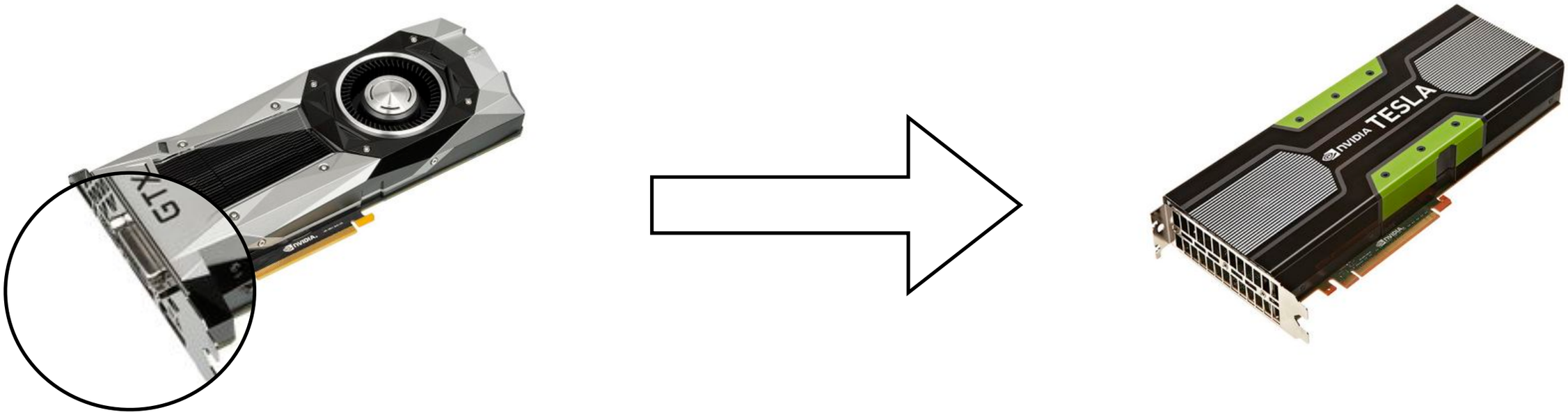

Theoretical GFLOP/s at base clock
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
2003
2005
2007
2009
2011
2013
2015


Theoretical Peak GB/s
GeForce GPU
Tesla GPU
Intel CPU
800
700
600
500
400
300
200
100
0
2003
2005
2007
2009
2011
2013
2015

GPU: Programming Model - Cuda

CUDA

- 2007 : CUDA (Compute Unified Device Architecture)
  - Unified graphics & compute architecture: Tesla, Fermi, Kepler, …, Volta
  - Programming model: C extensions
- Tesla Specific nvidia GPU for HPC



cuda overview

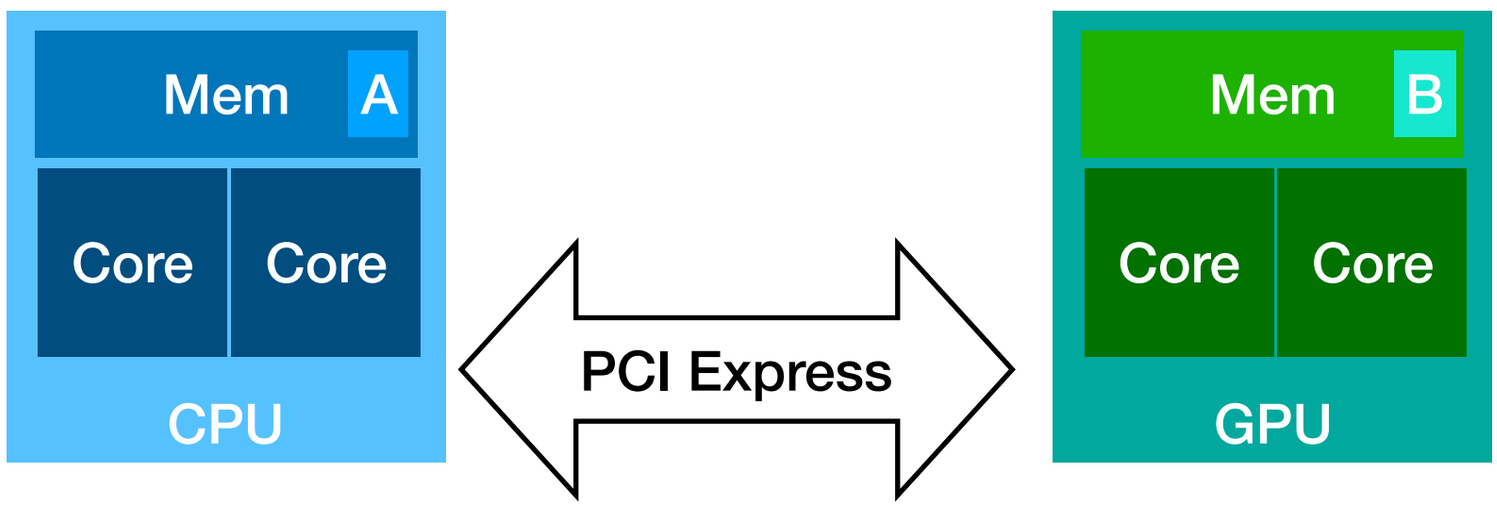
- For the demonstration we use the *deprecated* pyCuda

- Cuda allow to develop and compile source code to run **compute kernel** on GPU

How to compute on GPU?

- The main program run on the CPU (the host)
- How to compute on GPU (the device)?
- 5 steps:

    1. Allocate memory on the device
    2. Copy data from the Host to the Device (HtoD)
    3. Execute the kernel on the device
    4. Copy data from the Device to the Host (DtoH)
    5. Free memory on the device



How to compute on GPU?

- Getting started with PyCuda

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
```

- Preliminaries

  - The kernel compute a_i^2 for each a_i in a numpy array

```
import numpy
a = numpy.random.randn(4,4)
```

How to compute on GPU?

- Define the kernel (not running yet)

```
mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)
```

- Kernel is write using C-syntax
- In this kernel each GPU thread used compute only one a_i^2
- Little bit complicated code source
- Kernel not running yet but compiled and loaded on the device (the gpu)

How to compute on GPU?

1. Allocate memory

   - we have to know the data-size
   - we have to work with 32-bit float (on gpu)

```
a = a.astype(numpy.float32)
```

   - we use cuda.mem_alloc

```
a_gpu = cuda.mem_alloc(a.nbytes)
```

2. Copy array a from host to device (i.e. in a_gpu array)

```
cuda.memcpy_htod(a_gpu, a)
```

How to compute on GPU?

3. Run the kernel

   - Find a reference to our pycuda.driver.Function

```
func = mod.get_function("doublify")
```

   - And run it in a block of threads of size 4x4 (well dimensioned for our nd-array)

```
func(a_gpu, block=(4,4,1))
```

How to compute on GPU?

4. Copy the result from device (gpu) to host (cpu)

   - First we prepare memory to receive result

```
a_doubled =
numpy.empty_like(a)
```

   - Second we copy from device to host

```
cuda.memcpy_dtoh(a_doubled, a_gpu)
```

5. Finally we can free the device memory

```
a_gpu.free()
```