# Calcul à hautes performances : de la modélisation à l'implémentation

Camille Coti

LIPN, CNRS UMR 7030, SPC, Université Paris 13, France

*18 février 2019*

## Roadmap

## Roadmap

## Introduction to scientific computing

Today : first lecture of the class

- Next Monday : Vittoria Rezzonico (EPFL) and Nicolas Grenèche (Paris 13)

Today's material (slides + code) can be found on my webpage

- `www.lipn.fr/~coti/cours`

## Roadmap

## Parallel = several computing units

Can be **several computation nodes**

- Each controled by its own OS, have their own memory
- Interconnected by a (fast) network

Can be **several processors**

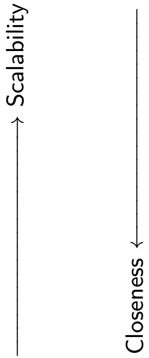
- Several processors on a motherboard
- Share the central memory
- Interconnected by the system bus

Can be several **cores**

- Several cores on a processor
- Some caches are shared, some are private

Can be several **instruction stream** (hyperthreading)

- Within a core
- ALU and caches are shared
- Each logical core has its own architectural states



## Parallel execution models

How does it run in parallel ? → *Flynn's taxonomy*

- **SISD** : Single Instruction, Single Data
  - Has some interest, but not what we are here for today
- **MIMD** : Multiple Instruction, Multiple Data
  - The general case in parallel computing : run different instruction streams on different data
- **SIMD** : Single Instruction, Multiple Data
  - Run the *same* instruction on different data
  - Vertor computing
- **MISD** : Multiple Instruction, Single Data
  - Very specific usage, mostly for redundancy, not relevant for today's talk

## MIMD : example

**Instructions**

| Step | P0 | P1 | P2 |
|---|---|---|---|
| 0 | int c ; | double d | double d ; |
| 1 | c = 2 ; | d = 3.0 ; | d = 5.0 ; |
| 2 | c++ ; | d /= 2.0 ; | d /= 2.0 ; |
| 3 | c%2 ; | d *= 4.0 ; | d += 1.0 ; |

**Process states**

| Step | P0 | P1 | P2 |
|---|---|---|---|
| 0 | int | double | double |
| 1 | 2 | 3.0 | 5.0 |
| 2 | 3 | 1.5 | 2.5 |
| 3 | 1 | 6.0 | 3.5 |

## SIMD : example

**Instructions**

| Step | Instruction |
|---|---|
| 0 | int c ; |
| 1 | c = getdata() ; |
| 2 | c++ ; |
| 3 | c%2 ; |

**Process states**

| Step | P0 | P1 | P2 |
|---|---|---|---|
| 0 | int | int | int |
| 1 | 2 | 3 | 8 |
| 2 | 3 | 4 | 9 |
| 3 | 1 | 0 | 1 |

## Current machines' architecture

Fast nodes

- **Multi-core** processors
- Several processors
- **Accelerators**

Specificities :

- Processors are slightly different from our desktop computers' CPU
  - Bigger **caches**
  - More cores
  - ECCmemory support
- Fast **interconnexion**
  - QuickPath Bus (Intel), HyperTransport (AMD)
  - Not always a unique bus : crossbar, multiples busses...

**Fast** network

- Ethernet : *out-of-band* communications
- Fast network : application
  - Low latency, high throughput
  - InfiniBand, Myrinet, proprietary networks (Tofu, Sea Star...)

## Hardware evolution

Explosion of the **number of cores**

- Several CPUs per node
- Advent of multi-core architectures
- What a core is is getting **blurrier**

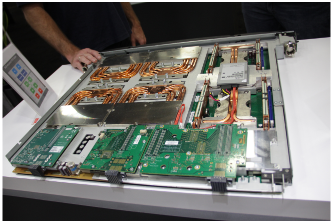
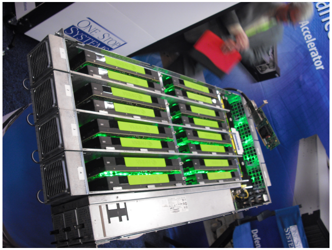
**Accelerators**

- GPU, Cell
- Xeon Phi
- Pezy-SC2

**Low latency** networks

- Myrinet, InfiniBand
- Order of magnitude w.r.t. moving data on the same node
  - Latency x2
  - Throughput / 2

Efforts on the **energy consumption**.

## Fastest machines in the world

**Top 500** : `https://www.top500.org`

- Runs a benchmark (LINPAK) that performs typical scientific computation operations
  - Factor and solve a dense linear algebra system of equations
  - Gaussian elimination with partal pivoting
- Used for statistics
  - Identify trends (architecture, size, network technology...)
  - By country, by OS...
  - Evolution over the years !
- First release in 1993, biannual (June at ISC, November at SC)

**Other ranking systems**

- HPCG (High Performance Conjugate Gradients) `http://www.hpcg-benchmark.org`
  - Krylov subspace solver
  - Additive Schwarz, symmetric Gauss-Seidel preconditioned conjugate gradient solver
  - Sparse linear system, mathematically similar to usual PDE problems
- Green500 `https://www.top500.org/green500`
  - Top500 data, energy efficiency
- Graph500 `https://graph500.org`
  - Computations on weighted, undirected graphs (search, shortest path...)
  - Relevant for 3D physics simulations, for example

## Top 500 : November 2018

- Rpeak is the *theoretical peak*
- Rmax is the *LINPACK performance*
- Rpeak and Rmax in TFlops/s, power in kW.

| Rank | System | Site | Cores | Rmax | Rpeak | Power |
|---|---|---|---|---|---|---|
| 1 | Summit | DoE / ORNL (USA) | 2,282,544 | 122,300.0 | 187,659.3 | 8,806 |
| 2 | Sunway TaihuLight | NSC Wuxi (China) | 10,649,600 | 93,014.6 | 125,435.9 | 15,371 |
| 3 | Sierra | DoE/LLNL (USA) | 1,572,480 | 71,610.0 | 119,193.6 | - |
| 4 | Tianhe-2A | NSC Guangzhou (China) | 4,981,760 | 61,444.5 | 100,678.7 | 18,482 |
| 5 | ABCI | AIST (Japan) | 391,680 | 19,880.0 | 32,576.6 | 1,649 |
| 6 | Piz Daint | CSCS (Switzerland) | 361,760 | 19,590.0 | 25,326.3 | 2,272 |
| 7 | Titan | DoE / ORNL (USA) | 560,640 | 17,590.0 | 27,112.5 | 8,209 |
| 8 | Sequoia | DoE / LLNL (USA) | 1,572,864 | 17,173.2 | 20,132.7 | 7,890 |
| 9 | Trinity | DoE / LANL / SNL (USA) | 979,968 | 14,137.3 | 43,902.6 | 3,844 |
| 10 | Cori | DoE/LBNL/NERSC (USA) | 622,336 | 14,014.7 | 27,880.7 | 3,939 |

**Summit** :

- IBM system, 4,608 nodes
- IBM POWER9 22C 3.07GHz processors (2/node)
- NVIDIA Volta V100s (6/node)
- Memory : 512GB DDR4 + 96GB HBM2 / node, 1600GB NV
- Dual-rail Mellanox EDR Infinibandnetwork
- `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`

## Former number 1's

| Name | Dates #1 | Nb cores | Rmax |
|---|---|---|---|
| CM-5 | 06/93 | 1 024 | 59.7 Gflops |
| Numerical Wind Tunnel | 11/93 | 140 | 124.2 Gflops |
| Intel XP/S 140 Paragon | 04/93 | 3 680 | 143.40 Gflops |
| Numerical Wind Tunnel | 11/94-12/95 | 167 | 170.0 Gflops |
| Hitachi SR2201 | 06/96 | 1 024 | 232.4 Gflops |
| CP-PACS | 11/96 | 2 048 | 368.20 Gflops |
| ASCI Red | 06/97 - 06/00 | 7 264 | 1.068 Tflops |
| ASCI White | 11/00 - 11-01 | 8 192 | 4.9 - 7.2 Tflops |
| Earth Simulator | 06/02 - 06/04 | 5 120 | 35.86 Tflops |
| BlueGene/L | 11/04 - 11/07 | 212 992 | 478.2 Tflops |
| Roadrunner | 06/08 - 06/09 | 129 600 | 1.026 - 1.105 Pflops |
| Jaguar | 11/09 - 06/10 | 224 162 | 1.759 Pflops |
| Tianhe-1A | 11/10 | 14 336 + 7 168 | 2.57 Pflops |
| K | 06/11 - 11/11 | 548 352 - 705 024 | 8.16 - 10.51 Pflops |
| Sequoia | 06/12 | 1 572 864 | 16.32 Pflops |
| Titan | 11/12 | 552 960 | 17.6 Pflops |
| Tianhe-2 | 6/13 - 11/15 | 3 120 000 | 33.9 Pflops |
| Sunway | 6/16 - 11/17 | 10 649 600 | 93.0 Pflops |
| Summit | 6/18 → | 2 282,544 | 122 300.0 |

## Access such machines

At **Paris 13 / USPC**

- **Magi** cluster
- Administrated and managed by Nicolas Grenèche (you will meet him next week) (awesome guy, very skilled, don't hesitate to ask for technical support)
- 50 compute nodes, 40 cores each
- 2 fat nodes, 512 GB memory each
- InfiniBand interconnect

National resources : **GENCI** (Grand Équipement National de Calcul Intensif)

- **TGCC** (Très Grand Centre de calcul du CEA)
  - Joliot-Curie : 6,8 petaflop/s + 2,1 petaflop/s, 382 TB
- **CINES** (Centre Informatique National de l'Enseignement Supérieur)
  - Occigen : 3,5 petaflop/s, 4 212 nodes, 85 824 cores, 283 TB
- **IDRIS** (Institut du développement et des ressources en informatique scientifique)
  - Jean Zay : 14 petaflop/s. "Scalar" part : 4.9 petaflop/s, 1528 nodes, 192 GB/node. "Converged" part : 9.02 petaflop/s, 261 nodes with 4 GPUs each

European resources : **PRACE**

## Roadmap

## How to choose how to program for a parallel machine ?

Look at the **architecture of the machine**

- Shared memory ?
- Vector processing unit(s) ?

Look at the **memory access patterns** of your application

- Regular ? Irregular ?
- All the processes at the same time ?
- Unexpected remote data accesses ?
- VERY big ?

## Cache blocking

Not *parallel* computing *per se*

- Work on your data by blocks that fit in caches
- Useful when the data is reused
  - *e.g. Matrix-matrix multiplication :* $O(n^2)$ *elements,* $O(n^3)$ *operations*

Blocked loop :

```
for( i = 0 ; i < size ; i+=block ) {
    for( j = 0 ; j < size ; j+=block ) {
        theblock1 = ( j+block < size ) ? block : (size-j) ;
        theblock2 = ( i+block < size ) ? block : (size-i) ;
        for( b = 0 ; b < theblock2 ; b++ ) {
            for( c = 0 ; c < theblock1 ; c++ ) {
                for( k = 0 ; k < size ; k++ ) {
                   /* ... */
                }
            }
        }
    }
}
```

Original loop :

```
    for( i = 0 ; i < size ; i++ ) {
      for( k = 0 ; k < size ; k++ ) {
          for( j = 0 ; j < size ; j++ ) {
             /* ... */
          }
      }
  }
```

## Cache blocking : example

Take `matmul.c`.

- Implements several **matrix-matrix multiplications** in $O(n^3)$ operations
- Same computation algorithm, different memory access patterns
  - Plain, naive pattern : `naiveMatMul()`
  - **Swapped loops** to make the inner loop work on consecutive data : `swappedMatMul()`
  - **Tiled** for cache blocking : `tiledMatMul()`
  - **Tiled** for cache blocking with **swapped loops** : `tiledSwappedMatMul()` and `tiledSwappedMatMul2()`

Compare the functions, compile and execute.

- Takes the matrix size as a parameter : `./matmul 256` to work on 256x256 matrices.

Compare the execution times.

If you have PAPI on your computer : use `matpul_papi.c`

- If PAPI cannot access your counters :

```
sudo bash -c "echo '-1' > /proc/sys/kernel/perf_event_paranoid"
```

Compare number of cache misses.

## Use SIMD registers

Your CPU has some **SIMD registers**

- Check the instruction set with `cat /proc/cpuinfo`

Vendor-specific

- Sorry AMD people, I will talk about Intel registers and instruction sets
- Some exist on AMD processors, some can be trivially transposed to AMD

Core idea :

- Stuff **several data words** in a **single register**
- **Execute instructions** on these registers

$\rightarrow$ Single instruction executed on several data

How many data words ?

- MMX : 64 b
- SSE and AVX : 128 b
- AVX2 : 256 b
- AVX-512 : 512 b

## Compiler-based vectorization

Modern compilers are already vetorizing whatever they can detect

- Loops on consecutive data...

Take `vector.c`. It is a very simple loop performing the same operation on consecutive data.

- Dump the assembler code with `gcc -S vector`
- Loop at the generated code and search for SIMD registers
- Try with different optimization options : -O0, -O3...

However, compilers cannot guess everything

- Take `matrix_novect.c`
- Dump the assembler code
- Look for vector operations
  - Reminder : double is on 64 bits, float is on 32 bits, xmm[0...15] registers contain 64 bits
- Change double for float, etc

## Manual vectorization

First possibility : **optimize your assembly code**

- Just use the SIMD registers like other registers, but put multiple things in them
- Dedicated load/store instructions
  - Example : MOVNTDQA is "Load Double Quadword Non-Temporal Aligned Hint"

Other possibility : **use intrinsics** in C and C++ code

- Registers : `__m256d`, `__m256i`, `__m256s`...
- Load operations : `toto = _mm256_loadu_pd( .... )`, `toto = _mm256_set_pd( .... )`....
- Store operations : `_mm256_storeu_pd( .... )`...
- Add, multiply...
- With FMA instructions : *Fused Multiply and Add*

Documentation :
`https://software.intel.com/sites/landingpage/IntrinsicsGuide`

## Intrinsics : example

**Example** : take `matrix_avx.c`

- Corresponds to `matrix_novect.c` with vectorization using intrinsics
- Loop : `i+=4` $\rightarrow$ 4 by 4
- `a` and `b` are **SIMD registers** that contain elements from the matrix
- `b` contains **contiguous data** : filled with `_mm256_loadu_pd()`
- `a` contains **non-contiguous data** : filled with `_mm256_set_pd()`
- `c` contains random-generated values
- The **computation** is made by `_mm256_add_pd()` and `_mm256_mul_pd()`
- The final result is **stored** by `_mm256_storeu_pd()`

**Compilation** :

- Option `-march=ative` is the easiest one
- Some other options : `-mavx`, `-mfma`...
- `#include <x86intrin.h>` gets you all the headers you need

## Intrinsics : exercise

**Exercise** : Implement a matrix-matrix multiplication using SIMD intrinsics

- Just start with the plain, simple pattern
- If available on your CPU, try FMA.

## Shared memory architecture

Particularity : the processing units **access some shared memory**

- Processing units ?
  - Sometimes threads, not always
  - Can also be processes

Low level of abstraction :

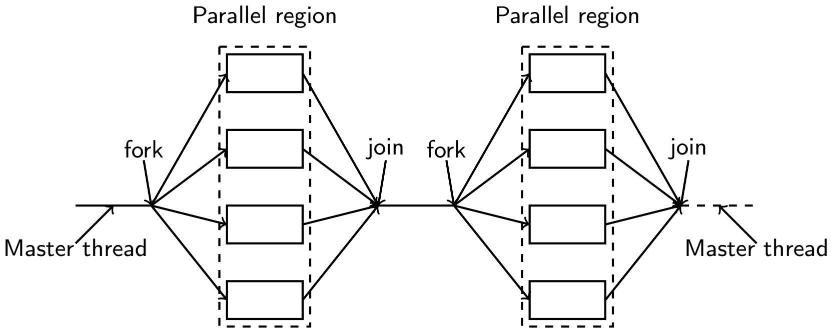
- Programming with **POSIX threads** (`pthread_create()`, `pthread_join()`...) or **processes** (created with `fork()` and `exec()`)
- Communication via segments of shared memory : POSIX `shm` framework, `Boost.Interconnect` segments...
- See for example `threads.c`

... But some tools exist to simplify thread management !

## OpenMP

### Annotation-based language

- Few modifications in the code
- Compilation directives
  - Start with `#pragma` : If the compiler does not support OpenMP, it is not enabled and the program works sequentially



A LOT of documentation can be found here :
`https://computing.llnl.gov/tutorials/openMP`

## What is OpenMP ?

Set of **functions, environment variables and compiler directives**

- High level programming
- The compiler is strongly involved

**OpenMP compiler**

- Uses compiler directives
- In charge with generating the threads, sharing work between threads, data location

**OpenMP library**

- Provides a run-time environment
- In charge with dynamic functions, at run-time

**Environment variables**

- Allows the user to set some parameters at run-time (number of threads...)
- In charge with everything specific for a given execution : hardware binding, stack size, etc

## Example : loop parallelization

### Global maximum on a table

**Algorithm 1:** Sequential computation of the maximum of a table

```
begin
    Data: Table of size N containing positive integers tab[]
    Result: Integer MAX
    $MAX = 0$;
    for $i \leftarrow 0$ to $N$ do
        if $tab[i] > MAX$ then $MAX = tab[i]$;
```

Parallelization of the `for` loop

- "Slice" the range on which the computation is made
- Slices are divided between threads

## OpenMP annotations

### Parallel sections

- `#pragma omp parallel` : beginning of a parallel section (fork)
- `#pragma omp for` : parallel for loop

### Synchronizations

- `#pragma omp critical` : critical section
- `#pragma omp barrier` : synchronization barrier

### Data visibility

- Private = visible only by this thread
- Shared = visible by all the threads
- By default :
  - Variables declared inside a parallel region is private
  - Variables declared outside are shared

```
#pragma omp parallel private (tid) shared (result)
```

## Compilation and execution

**Headers** `#include <omp.h>`
**Compilation** Enable OpenMP with an option of the compiler

- For gcc : `-fopenmp`

Reminder : if the option is not enabled, the annotations are ignored (not the functions).
**Execution** Number of threads :

- By default : the environment discovers how many cores are available and uses them all
- Set by the user using the environment visible `$OMP_NUM_THREADS`

## OpenMP parallel region

The **parallel region** is declared using

```
#pragma omp parallel
```

Every thread executes what is inside of the **structured block**

- Warning : the opening brace must be at the beginning of the line
- Branching (*e.g. goto) to the inside or the outside of a parallel region are forbidden*

### Hello World 0.1

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
    printf("Hello from outside\n" );
#pragma omp parallel
    {
      printf("Hello World !\n" );
     }
    return EXIT_SUCCESS;
}
```

## Variable scope

Defining **variable scope**

```
#pragma omp parallel private ( tid, numthreads )
#pragma omp parallel private ( a, b ) shared ( c, d )
```

Threads are identified by their **rank** :

- Thread number : `omp_get_thread_num()`
- Number of threads in the parallel program : `omp_get_num_threads()`
  - Can be set by the `OMP_NUM_THREADS` environment variable and the `omp_set_num_threads()` function

## Using a shared variable

### Hello World 2.0

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(){
    int numthreads, tid;
#pragma omp parallel private( tid ) shared ( numthreads )
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        if( 0 == tid ) {
           numthreads = omp_get_num_threads();
        }
    }
    printf("Number of threads = %d\n", numthreads);
    return EXIT_SUCCESS;
}
```

Warning : pay attention to mutual exclusion on shared variables

- Here : only one threads writes in the shared variable, which is read only and the end of the exeuction of all the threads
- Several ways to control mutual exclusion and causal ordering between operations

## Mutual exclusion and synchronization

Ensure **mutual exclusion** :

- Defining critial sections : constructor `critical`
- Locks : `type omp_lock_t`
- Atomicity : constructor `atomic`

Differences :

- Useability, syntax
- Restrictions : exceptions, how can we get out of it...
- Ease of use, likelyhood to write bugs

**Synchronization** between threads :

- *Explicit* barrier : `#pragma omp barrier`
- The compiler adds *implicit* barriers : end of a parallel region, end of a loop, end of a `single` region...

## SIMD in OpenMP

OpenMP has **some SIMD extensions**

```
#pragma omp simd
 for(int n=0; n<8; ++n) an += bn;
```

Can be passed a clause :

- Collapse, reduction....
- Information about the scope of the variables...

## Tasks in OpenMP

**Task-oriented** paralelism :

- Pieces of computation that are independant from each other
- A task is executed by a thread
- If no thread is available : the task is queued and executed later

**How to use this model** : create the tasks, they are executed by threads from the pool

- Independant computations
- Can be recursive

Adn wait for the end of their execution.

```
#pragma omp task
```

**Synchronisstion** : wait for the end of the tasks

```
#pragma omp taskwait
```

## Example : Fibonacci

Warning : this example is meant only for education purpose, it gives very poor performance.

- Very few computations, a lot of interactions between the threads

```
int  fib(int n){
  int i, j;
  if ( n < 2 ) return n;
  else {
#pragma omp task shared(i) firstprivate(n)
       i = fib(n-1);
#pragma omp task shared(j) firstprivate(n)
       j = fib(n-2);
#pragma omp taskwait
       return i+j;
    }
}
```

Initial call :

```
#pragma omp parallel shared(n)
  {
#pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
```

## Data dependency

Shared variables between tasks

- As usual, `shared`, `firstprivate`, `lastprivate`
- Watch for shared vatiables
    - Critical sections, etc
- The returned value can be used

**Data dependency** between threads can be defined

```
 for (int i = 0; i < T; ++i) {
#pragma omp task shared(x, ...) depend( out: x) // T1
    foo(...);
#pragma omp task shared(x, ...) depend( in: x) // T2
    bar(...);
#pragma omp task shared(x, ...) depend( in: x) // T3
    toto(...);
}
```

- T1 $\prec$ T2, T3
- T2 // T3

From these informations, the run-time environment **buids a DAG** and **schedules** the tasks.

## Using OpenMP to program on GPU

OpenMP can be used with some extensions to **program GPUs** : the **OmpSs** model

```
#pragma omp target device ({ smp | cuda })
```

More information : `https://pm.bsc.es/ompss`

## Programming on GPUs

GPUs are **great**

- A lot of processing units
- High bandwidth local memory

... but GPUs have **some restrictions**

- Mostly **vector-based** computation
- Need to move data back and forth between the host and the device
- Slow double precision computations

Can be programmed using

- Cuda
- OpenCL
- SYCL...

## Architecture

On a GPU, a **core** has a very specific architecture

- One instruction stream
- Multiple ALUs

Consequence : all the threads on a core must **execute the same instruction**

- Conditional branches are executed sequentally
- No gain...

Therefore, **performant on vector-like computation patterns**

## Cuda : example

Example provided by Nvidia's documentation : take `add.cu`

**Unified memory** can be accessed from the GPU or the CPU

- Allocated by `cudaMallocManaged`
- Freeed by `cudaFree`

The function to execute on the GPU is called a **kernel**

- Definition starts with `__global__`
- Vectorized automatically or manually
- Started by `add«.....»` : provides the device number

## StarPU

What is StarPU

- Task-based execution system
- Write tasks (defined by "codelets") providing **data dependencies** between tasks
- StarPU **infers the DAG** (statically or at run-time)
- StaPU **schedules** the tasks on the available resources, *i.e.* **on the CPU and the GPUs**.

A **codelet** describes a computation kernel

- A **task** is the application of a codelet on data

## Defining a codelet

Two parts :

- Defining **the kernel** itself

The prototype *must* be as follows :

```
void cpu_func(void *buffers, void *cl_arg)
{
    printf("Hello world\n");
}
```

- Defining **the codelet**

Provide information on the kernel, its in/out buffers....

```
struct starpu_codelet cl =
{
    .cpu_funcs = { cpu_func },
    .nbuffers = 0
};
```

## Submitting the task

To execute a task :

- **Create** the task and set the codelet

```
struct starpu_task *task = starpu_task_create();
task->cl = &cl;
```

- **Submit it** to the StarPU scheduling system

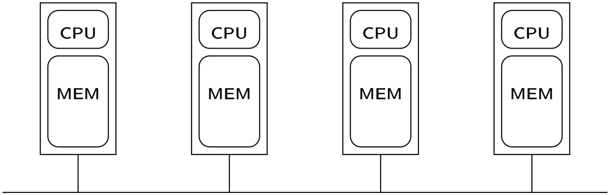
```
starpu_task_submit(task);
```

More documentation at :
`http://starpu.gforge.inria.fr/doc/html/index.html`

## Distributed memory

The programmer is in charge with **data locality**

- Each process has its **own memory space**
- **Explicit** data movements
- Need an **communication library** : set of functions, routines... to move data and handle information about the processes' organization
- And a **run-time environment** to start the parallel processes on the distributed resources and orchestrate the resources



## Message Passing Interface

Considered as the *de facto* interface for programming parallel distributed programs

- Huge set of routines

### History

- First call for contributions : SC 1992
- 1994 : MPI 1.0
  - Basic point-to-point communications
  - collective communications
- 1995 : MPI 1.1 (clarifications)
- 1997 : MPI 1.2 (clarifications and corrections)
- 1998 : MPI 2.0
  - Dynamic process management
  - RDMA
- 2008 : MPI 2.1 (clarifications)
- 2009 : MPI 2.2 (corrections, few additions)
- MPI-3.0 (September 2012) and MPI-3.1 (June 2015).
  - Non-blocking collective operations
  - MPI SHM
  - a LOT of other sections

## Process naming system

Processes that communicate together belong to the same **communicator** :

- All the processes are in MPI_COMM_WORLD
- Everyone is alone in its own MPI_COMM_SELF
- MPI_COMM_NULL does not contain any process

Other communicators can be created at run-time

Processes are designated by their **rank**

- Unique in a given communicator
  - Rank in MPI_COMM_WORLD = absolute rank in the application
- Used to send/receive messages

## Deployment of the application

**Start** `mpiexec` starts the processes on the remote machines

- Start = execution of a program on a remote machine
  - The binary executable must be accessible on the remote machine
- Can execute a different binary depending on process ranks
  - "True" MPMD
- Command-line parameters are transmitted

Input / outputs / signals are **forwarded**

- `stderr, stdout, stdin` are forwarded to the start-up process (mpiexec)
- MPI-IO for I/O

**Finalization**

- `mpiexec` returns when all the processes are done
- Or when one process has exited abnormally (crash, failure...)

Using a **batch scheduler**

- Everything is done by the batch scheduler
- Start the application, outputs in files...

## Communication model

### Asynchronous

- Communication time : finite, unbounded

### Communication modes

- Small messages : **eager**
  - The sender sends the message on the network and returns as soon as the message is transferred to the network layer
  - If the receiver is not in a receive call, the message is bufferized
  - When the receiver enters a receive call, it starts looking in its buffers, to check whether the message is already here
- Big messages : *rendez-vous*
  - The sender and the receiver must be in the communication call
  - Rendez-vous mechanism :
    - Send a small fragment
    - The receiver acknowledges
    - Send the rest of the message
  - The sender returns only once it has sent all the message. The receiver is receiving the message : no bufferization.

## Hello world in MPI

**Initialization** of the MPI library

- `MPI_Init( &argc, &argv );`

**Finalization**

- `MPI_Finalize( );`

If a process exits before `MPI_Finalize( );`, it will be considered as an abnormal exit

*These two functions are MANDATORY !!*

**How many processes** are there on the application ?

- `MPI_Comm_size( MPI_COMM_WORLD, &size );`

What is my **rank** ?

- `MPI_Comm_rank( MPI_COMM_WORLD, &rank );`

## Hello World in MPI

### Full code

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main( int argc, char** argv ) {
  int size, rank;

  MPI_Init( &argc, &argv );

  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  fprintf( stdout, "Hello, I am rank %d in %d\n",
                    rank, size );

  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

## Hello World en MPI

**Compiler** : `mpicc`

- Wrapper around the C compiler of the system
- Provides paths to `mpi.h` and the MPI library
- Roughly equivalent to
  `gcc -L/path/to/mpi/lib -lmpi -I/path/to/mpi/include`

```
mpicc -o helloworld helloworld.c
```

**Execution** with `mpiexec`

- Provide a list of hosts in a (`machinefile`)
- Number of processes to start

```
mpiexec -machinefile ./machinefile -n 4 ./helloworld
Hello, I am rank 1 in 4
Hello, I am rank 2 in 4
Hello, I am rank 0 in 4
Hello, I am rank 3 in 4
```

## About Python

Python bindings exist, but are **non-official**

- Not part of the standard
- For instance : mpi4py (high quality)

```
from mpi4py import MPI
```

The Python script still needs an interpreter :

```
$ mpiexec -n 8 python helloWorld.py
```

**Particularity with Python** : neither `MPI_Init` nor `MPI_Finalize`

## Example in Python

Communicators : `MPI.COMM_WORLD`, `MPI.COMM_SELF`, `MPI.COMM_NULL`

```
  #!/bin/python

from mpi4py import MPI

def main():
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    print "hello from " + str( rank ) + " in " + str( size )


if __name__ == "__main__":
    main()
```

## Peer-to-peer communications

Two-sided communications can be :

- **Blocking** : MPI_Send, MPI_Recv
- **Buffered** : MPI_Bsend, MPI_Brecv
- **Non-blocking** : MPI_Isend, MPI_Irecv
- **Buffered, non-blocking** : MPI_Ibsend, MPI_Ibrecv
- **Asynchronous** : MPI_Asend, MPI_Arecv
- Returns **only if the matching receive has been posted** : MPI_Ssend
- Can be used **only if the matching receive has been posted** : MPI_Rsend

One-sided communications :

- MPI_Put, MPI_Get
- Asynchronous, non-blocking

## Communications

### Data

- buff : send / receive buffer
- count : number of elements, of type `datatype`
- datatype : type of the communicated data
  - Use MPI data types
  - Ensures portability (including 32/64 bits, heterogeneous environments...)
  - Standard data types, new ones can be defined (derived data types)

### Process identification

- Use the couple communicator / rank
- Reception : can use a **wildcard**
  - `MPI_ANY_SOURCE`
  - After completion of the reception, the sender can be found in the status

### Communication identification

- Use a tag
- Reception : can use a **wildcard**
  - `MPI_ANY_TAG`
  - After completion of the reception, the tag can be found in the status

## Ping-pong

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main( int argc, char** argv ) {
  int rank;
  int token = 42;
  MPI_Status status;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
  } else {
    if( 1 == rank ) {
      MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
      MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD );
    }
  }
  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

## Ping-pong

### Remarks

- For each send, there is **always** a matching receive
  - Same communicator, same tag
  - Sender's rank and receiver's rank
- Rank used to determine what need to do
- Integers are sent : → MPI_INT

### Frequent mistakes

- The data type and number of elements must be the same in the send and the receive calls
  - The receiver expects to receive what was sent
- Matching MPI_Send et MPI_Recv
  - Two MPI_Send or two MPI_Recv = deadlock !

## Ping-pong illustated

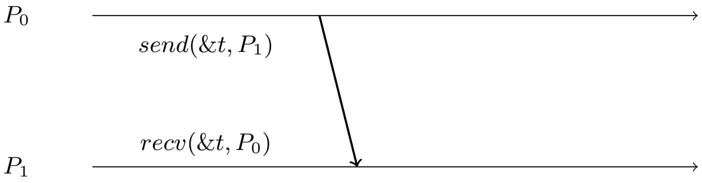
- Rank 0 sends a token
- Rank 1 receives it and sends it back to rank 0
- Rank 0 receives it.

```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

## Ping-pong illustated

- Rank 0 sends a token
- Rank 1 receives it and sends it back to rank 0
- Rank 0 receives it.

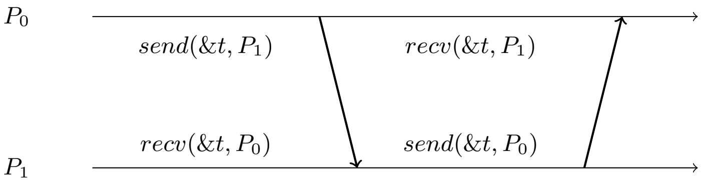
```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



## Ping-pong illustated

- Rank 0 sends a token
- Rank 1 receives it and sends it back to rank 0
- Rank 0 receives it.

```
if( 0 == rank ) {
    MPI_Send( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv( &token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status );
} else if( 1 == rank ) {
    MPI_Recv( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    MPI_Send( &token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```



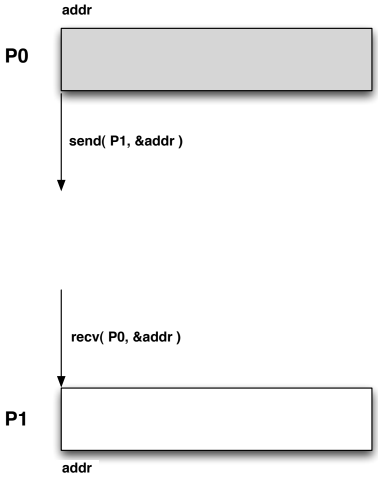
## How communications are actually made

Every process (sender or receiver) has a **buffer** corresponding to the message

- Memory **must be allocated** both on sender's side and receiver's side
- We do not send more elements than the available space

Data must be linearized (marshalled) in the buffer

- We send a buffer, a table of elements, a row of bytes...



## Some communication patterns : master-workers

**Data distribution** : the master distributes input data to the workers

- The master demultiplexes the input data, multiplexes the results
- The slaves **do not communicate** with each other

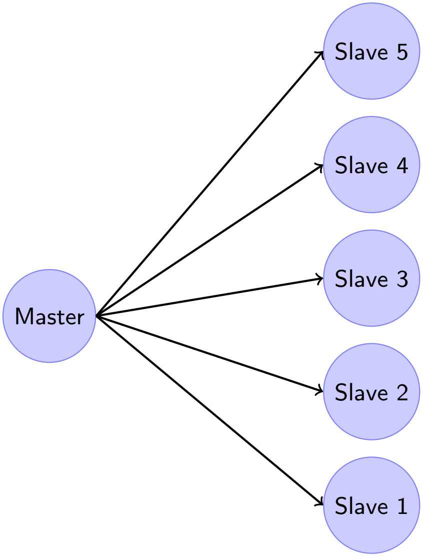
**Efficiency** : the master manages queues for the input data and the results

- Sequential part of the computation
- Communications : master $\leftrightarrow$ slaves
- The slaves do not work when they are waiting for data or when they are sending their results

The slaves only take actual part of the computation

- Can give a good speedup at large scale (slaves $>>$ master) *if they do not communicate often*
- Not very efficient with only a few processes
- Might cause a bottleneck on the master

## Master-slave



**Static** load balancing :

- Data distributed using `MPI_Scatter`
- Results gathered using `MPI_Gather`

Example : masterworker3.c
**Dynamic** load balancing :

- *Pull* mode : the slaves ask for work
- The master sends chunks one by one

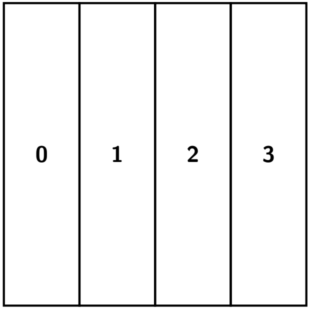
Example : masterworker1.c, masterworker2.c

## Some communication patterns : domain decomposition

**Process grid** : the data is sliced, a process is

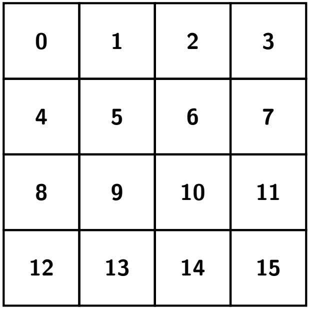
### 1D decomposition

The data is sliced in bands



### 2D decomposition

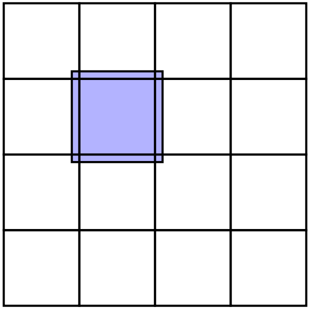
The data is sliced in rectangles : more scalable



## Ghost region

### Boarder between sub-domains

- An algorithm can need values of neighboring points to compute the new value of a point
  - Image processing (gradient...), cellular automata...
- Replicate data around the border
  - Each process keeps a bit of data from the neighbors
  - Updated at the end of the computation



## Exercise : data exchange between neighbors

Write a parallel program using MPI that :

- Initializes a matrix on each process
- Exchange a ghost region

Using a 1D and a 2D decomposition.

You can start from `cart_comm.c` : the program extracts communicators from a 2D process grid.

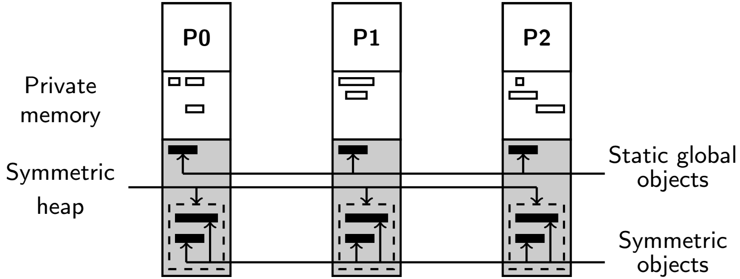
## OpenSHMEM

Other **communication and memory model**

- Shared heap
- **One-sided communications**

Memory model : **symmetric heap**

- Private memory vs shared memory (heap)
- Memory allocation in the shared heap is a *collective communication*



## OpenSHMEM : Example

Allocation in the shared heap :

- `shmalloc` function
- Warning : collective

Data movements :

- Fonctions `shmem_*_put`, `shmem_*_get`
- One function for each data type

```
short* ptr = (short*)shmalloc( 10 * sizeof( short ) );
if (_my_pe() == 0) {
    shmem_long_put( ptr, source, 10, 1 );
}
```

## Global Address Space

Concept of **global address space** :

- Program distributed memory just like shared memory
- Participation from the **compiler**
- The union of the distributed memories is seen by the programmer **as a shared memory**

In practice :

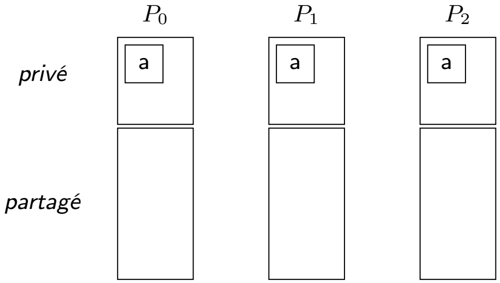
- The programmer declares the **visibility** of his/her variables : private (by default) or **shared**
- Arrays : The programmer declares the size of the blocks that will be placed on each process
- The compiler is in charge with :
  - **Distributing the shared variables** in the memory of the processes
  - **Translating remote accesses** $(a = b)$ into communications

Issues related to the fact that the memory is distributed **are not seen** by the programmer.

## Examples

**PGAS** languages :

- Unified Parallel C (UPC), Titanium, CoArray Fortran



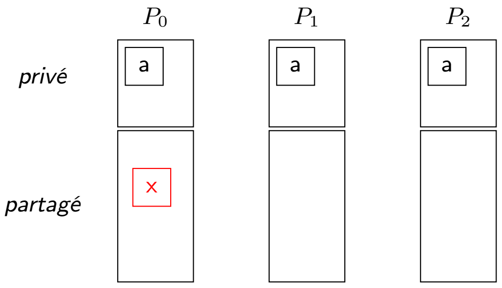
int a ;

shared int x ;

## Examples

**PGAS** languages :

- Unified Parallel C (UPC), Titanium, CoArray Fortran



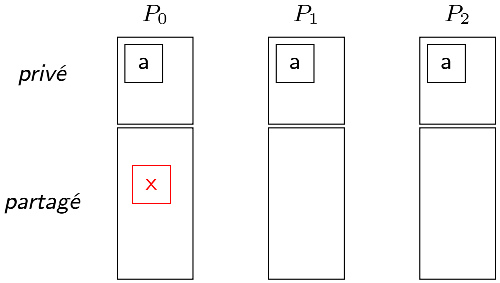
int a ;

shared int x ;

## Examples

**PGAS** languages :

- Unified Parallel C (UPC), Titanium, CoArray Fortran

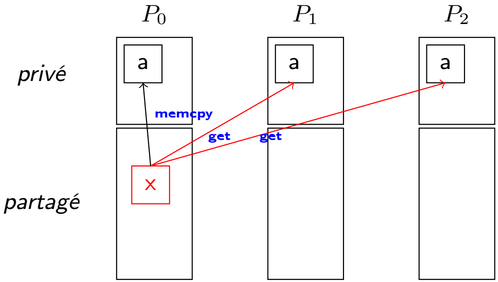


```
int a;
shared int x;
int a = x;
```

## Examples

**PGAS** languages :

- Unified Parallel C (UPC), Titanium, CoArray Fortran



```
int a;
shared int x;
int a = x;
```

## UPC : Example

Example :

- A variable `x` is shared, and therefore accessible from all the processes
  - The compiler will place it in the memory of a process of its choice.
- Process 0 (called `thread` in UPC terminology) initializes it to 42.
- A global barrier makes sure that all the processes have reached this point of the program.
- All the processes read the value of `x` and put it into a private variable of their own.
  - The compiler generates inter-process network communications (in all likelihood *get*)

```
shared int x;
int a;
if( 0 == MYTHREAD ) {
    x = 42;
}
upc_barrier;
a = x;
```

## Roadmap

## A few words on performance evaluation

### Speed-up

-

### Sequential application profiling

- PAPI : Performance API
- Hardware counters
- Counts operations, cache hits/misses, erroneous branch predictions...
- `http://icl.utk.edu/papi/`

### General profiling

- VTune
- A lot of information, including vector performance
- `https://software.intel.com/en-us/vtune`

### Parallel applications profiling

- Tau : profiling and tracing `http://tau.uoregon.edu`
- EZtrace : modular `http://eztrace.gforge.inria.fr`
- mpiP : lightweight, time spent in MPI routines `http://mpip.sourceforge.net`

## Roadmap