RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

Programming parallel computers

Introduction

Xavier JUVIGNY, AKOU, DAAA, ONERA
xavier.juvigny@onera.fr

Course Parallel Programming
- January 15th 2024 -

[1] ONERA, [2] DAAA

Table of contents

1. Motivations
2. Classification of parallel architectures

Overview

1 Motivations

2 Classification of parallel architectures

Parallel architecture

The main story

- Processors with multiple computing cores for faster computation
- Simultaneously using many cores for a unique application
- Performance benchmark in scientific computing is given by the number of FLoating Operations Per Second (FLOPS)

Hardware implementation

- Many computing cores sharing the same main memory inside a computer
- Using many computers linked with a fast, specialized Ethernet connection
- Mixing both technologies mentioned above.

Interests of parallel architecture ?

- **Gordon Moore's "Law"** : In 1965, Gordon Moore (one of Intel's founder) observed that the number of transistors for each generation of processors doubled in eighteen months, thereby doubling the computing power.
- In fact, it isn't a law, but it was used by processor builders as a roadmap until 2000 years to increase the frequency of computing cores.
- **Limitations of Gordon Moore's Law** : The miniaturisation of transistors and the increase in their frequencies lead to higher heat production inside the processor. Moreover, miniaturisation is now at the molecular scale, and one must consider quantum effects (such as tunnelling) when manufacturing a processor.
- Nowadays, **Moore's law is still verified**, not by doubling the number of transistors inside a computing core, but rather by increasing the number of computing cores inside a processor or computer.

Parallel computing example (1)

Control command

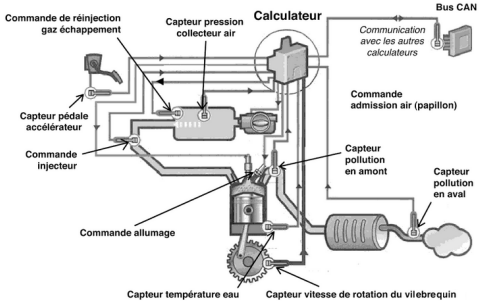


Figure – Car's control command

- Many tiny computers are specialized for specific tasks : ABS, motor optimization, lighting, air conditioning, wheel pressure optimization, fuel/air mixture, battery optimization, and so on.
- Computations must be completed within specific time constraints.
- Many parameters are interdependent (external air temperature, wheel pressure, optimal speed and oil consumption).

Parallel computing example (1)

Control command (continuation)

Figure – None Control Command of a reactor

- Another control command : managing nuclear power reactors
- High real-time constraint algorithms
- Many complex computations
- One computing core isn't sufficient to meet tight real-time constraints.
- **Solution** : Concurrent execution of interdependent tasks on multiple computing cores.

Parallel computation example (2)

Physical Simulation

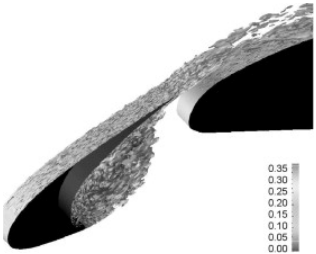


Figure – Turbulent noise generated by a plane's slate wing

- Turbulence : very small phenomena (millimeter scale). Requires a mesh with lots of tiny triangles.
- Typically, the mesh must contain five to ten billions vertices with five unknown variables for each vertex.
- Minimum memory requirement : 7 TB (terabytes).
- Sequential computation time : 23 days to simulate 10^{-2} seconds.

Parallel computation example (3)

Artificial intelligence

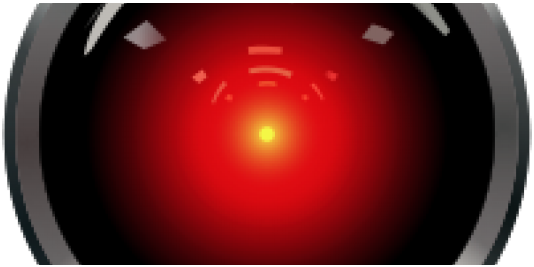


Figure – A very famous artificial intelligence (HAL)

- Deep learning is used in AI for tasks such as categorizing pictures, automatic translations, detecting cancerous cells, and autonomous vehicles.
- In a sequential process, it requires more than a year to learn
- With GPGPU, it takes about a few hours or days
- March 2016 : AlphaGo wins against the world GO human champion (supervised learning).
- October 2017 : AlphaGo zero wins against alphaGo with a score of 100 games to zero (deep learning).

Parallel computation (4)

Picture treatment

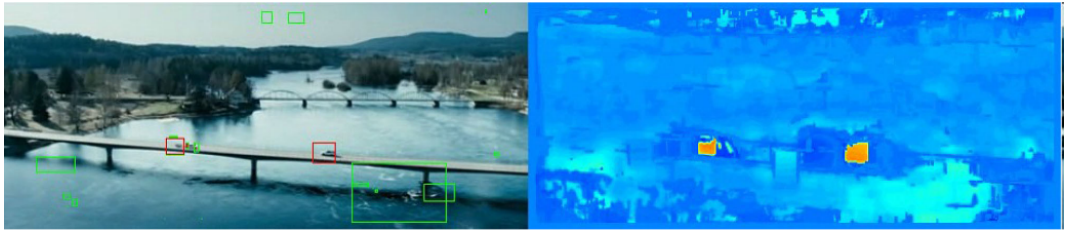


Figure – Real-time constraint treatment of a video with 30 frames/s (resolution 1920 $\times$ 1080 pixels)

- Required for optical sensors for navigation of autonomous vehicles, for super-resolution pictures derived from low-resolution video, and more.
- Involves a significant amount of computations (PDE equations to solve).
- Requires the use of GPGPU and parallel algorithms to meet real-time constraints.

Top 10 of supercomputers (June 2020)

Name	Core	Perf. (TFlops/s)	Constructor	Country	Power (kW)
Fugaku	7 299 072	415 530	Fujitsu	Japan	28 335
Summit	2 414 592	148 600	IBM	USA	10 096
Sierra	1 572 480	94 640	IBM/NVidia	USA	7 438
Sunway TaihuLight	10 649 600	93 014	NRCPC	China	15 371
Tianhe-2A	4 981 760	61 444	NUDT	China	18 482
HPC5	669 760	35 450	Dell EMC	Italy	2 252
Selene	277 760	27 580	Nvidia	USA	1 344
Frontera	448 448	23 516	Dell EMC	USA	?
Marconi-100	347 776	21 640	IBM	Italy	1 476
Frontier	591 872	1 102	HPE	USA	21 000

Remark : Nowadays, we look for Flops/Watt performances

Top 10 of supercomputers (November 2022)

Name	Core	Perf. (TFlops/s)	Constructor	Country	Power (kW)
Frontier	8 730 112	1 102 000	HPE	USA	21 100
Fugaku	7 630 848	442 010	Fujitsu	Japan	29 899
LUMI	2 220 288	309 100	HPE	Finland	6 015
Leonardo	1 463 616	174 700	Atos	Italy	5 610
Summit	2 414 592	148 600	IBM	USA	10 096
Sierra	1 572 480	94 640	IBM/NVidia	USA	7 438
Sunway TaihuLight	10 649 600	93 010	NRCPC	China	15 371
Perlmutter	761 856	70 870	HPE	USA	2 589
Selene	555 520	63 460	Nvidia	USA	2 646
Tianhe-2A	4 981 760	61 440	NUDT	China	18 482

Remark : Nowadays, we look for Flops/Watt performances

Overview

1 Motivations

2 Classification of parallel architectures

Shared memory architecture

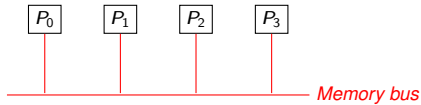


Figure – Simplified scheme of shared memory parallel architecture

Many computing cores share the same main memory.

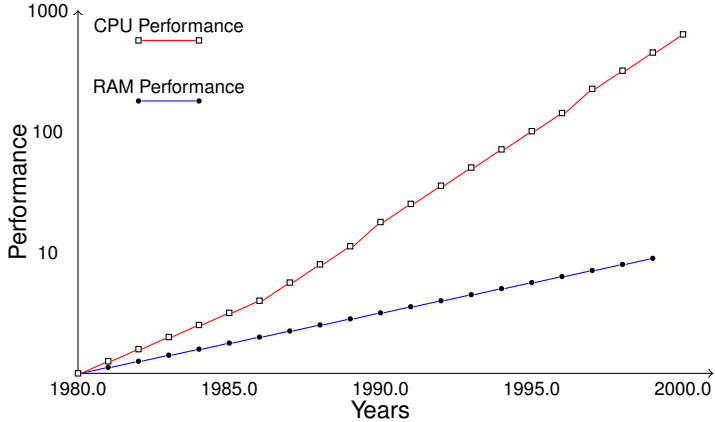
Examples

- Recent multi-core processors
- Graphics cards with 3D acceleration
- Phones, tablets, etc.

Memory access problem

- Optimization of memory access
- Simultaneous read/write accesses to the same memory location

Memory access



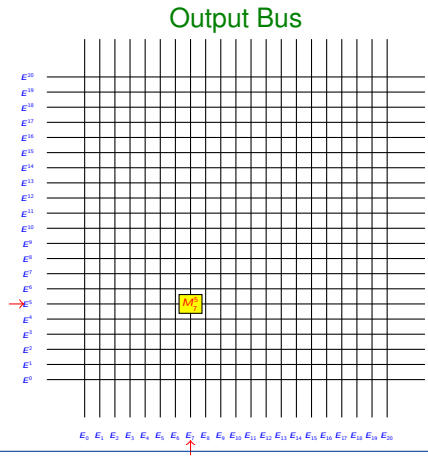
Latency memory example (Haswell architecture)

Level	Size	Latency (cycles)	Physical location
L1 Cache	16/16 ko	4	In each core
L2 Cache	256 ko	12	Shared by two cores
L3 Cache	12 Mo	21	Shared by all cores
Ram	32 Go	117	SRAM on mother board
Swap	100+ Go	10 000	Hard disk or SSD

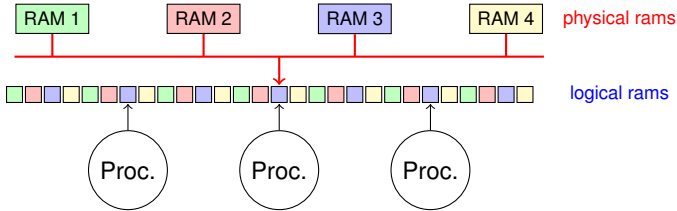
Conclusion

- Memory is becoming slower in comparison to the instruction execution of the processor.
- This discrepancy is exacerbated in multicore architectures.

How does RAM work ?



Interleaved RAMs

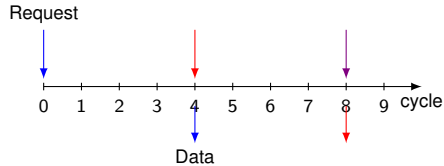


Interleaved memory

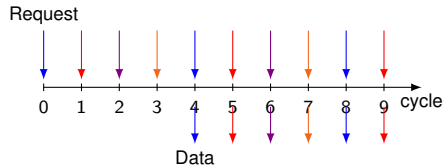
- Multiple physical memory units interleaved by the memory bus
- Number of physical memory units $\equiv$ number of ways
- Number of contiguous bytes in a single physical memory unit $\equiv$ width of way
- Quadratic cost in € to build, relative to number of memory units !

Interleaved memory access

Classic memory access



Four ways interleaved memory access



Cache memory

Consequences of grid architecture for RAMS

The larger the memory, the larger its grid, resulting in slower read and write access.

Cache memory

- Fast small memory unit used for storing temporary data
- When there are multiple accesses to the same variable in a short time, it speeds up read or write access
- Cache memory managed by the CPU (although cache memory for GPUs can be managed by the programmer)
- **Consequence** : to optimize their program, the programmer must be aware of the strategies used by the CPU.

Cache memory

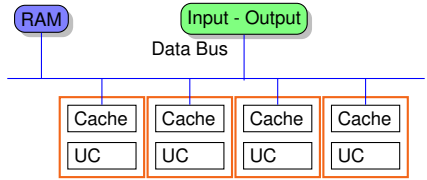
CPU Strategy

- **Cache Line** : store contiguous memory variables in cache (typically 64 bytes on Intel processor)
- **Associative memory cache** : each cache memory address mapped to fixed RAM address (using modulo)

Consequences

- advantageous to have contiguous access in memory.
- preferable to use data stored in the cache as soon as possible
- Spatial and temporal localization of data

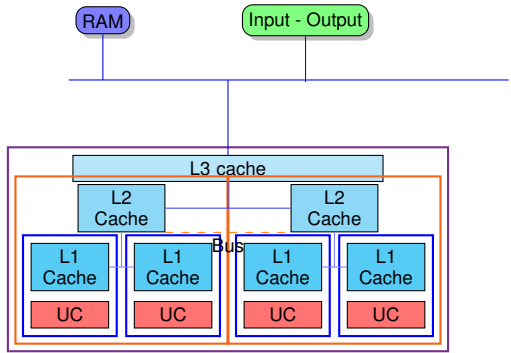
Memory organization on multi-processor computer



Data coherence between memory caches :

- In a unique cache, the datum's value is valid, and no synchronization is needed.
- When a datum is shared with another memory cache, each access involves verifying if the datum has been modified by another core. The cache writes the datum as invalid when modifying its value.
- If the value in the cache is modified, the corresponding value in RAM is now invalid. The cache updates the value in RAM if another core reads the value.
- When the value is invalid in the cache, the next read of this value must access the value in RAM.

Many cores cache organization



Same issue arises with cache consistency, but there is a need for coherence of data between cache levels. The complexity increases with the number of cache levels.

Tools for shared memory computation

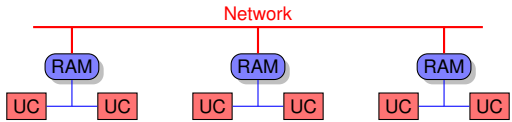
Many tools can be used to implement multiple "threads" and synchronization in memory. The most commonly used ones are :

- OpenMP : Compilation directives and a small C API (`#pragma`).
- Standard Library in C++ (threads, asynchronous functions, execution policies).
- oneTBB (oneAPI Threading Building Block library, Intel) : An open-source library from Intel.

However, the programmer must be cautious about memory conflict access :

- When a thread writes to a datum, and other threads simultaneously read the same datum.
- When multiple threads write to the same datum.
- The programmer should not rely on the instruction order in the program due to out-of-order optimizations by the compiler and processor.

Distributed memory



- Each computing unit can read/write to local RAM ; the set containing the computing unit and the RAM is called the **Computing Node**.
- Data is exchanged between computing nodes through a specialized bus or a specific Ethernet link.
- On an Ethernet link, it is the responsibility of the programmer to explicitly exchange data between computing nodes.
- Specific efficient algorithms and libraries are required.
- It is possible to compute on many thousands of computing cores.
- The limitation is only imposed by electricity consumption (with linear cost).

Distributed parallelism context

All libraries managing distributed parallel computation provide similar functionalities.

Running a Distributed Parallel Application

- The user is provided with an application to run on a specified number (`nbp`) of computing nodes (given during application execution).
- The computing nodes where the application is launched are defined either by default or in a file provided by the user.
- The default output for all processes is the terminal output from which the application was launched.
- A communicator (defining a set of processes) is set by default, including all launched processes (`MPI_COMM_WORLD`).
- The application assigns a unique number to each process in a communicator (numbering from 0 to `nbp`-1).
- All processes terminate the program simultaneously.

Managing the context in your program

- Call the initialization of the parallel context before using other functions in the library (`MPI_Init`).
- Obtain the number of processes contained in the communicator (`MPI_Comm_size`).
- Retrieve the rank of the process within the communicator (`MPI_Comm_rank`).
- After calling the last library function, invoke the termination of the parallel context to synchronize processes (`MPI_Finalize`). Failure to do so may result in program crashes.

```
#include <mpi.h>
int main(int nargs, char const* argv[])
{
    MPI_Comm commGlob;
    int nbp, rank;
    MPI_Init(&nargs, &argv);// Initialization of the parallel context
    MPI_Comm_dup(MPI_COMM_WORLD, &commGlob);// Copy global communicator in own communicator;
    MPI_Comm_size(commGlob, &nbp);// Get the number of processes launched by the used;
    MPI_Comm_rank(commGlob, &rank);// Get the rank of the process in the communicator commGlob.
    ...
    MPI_Finalize();// Terminates the parallel context
}
```

Point to point data exchange

Constitution of a Data Message to Send

- The communicator used to send the data.
- The memory address of the contiguous data to be sent.
- The number of data to send.
- The type of the data (integer, real, user-defined type, etc.).
- The rank of the destination process.
- A tag number to identify the message.

Constitution of a Data Message to Receive

- The communicator used to receive the data.
- A memory address of a buffer to store the received data.
- The number of data to receive.
- The type of the data (integer, real, user-defined type, etc.).
- The rank of the sender process (can be any process).
- A tag number to identify the message (can be any tag if needed).
- Status of the message (receive status, error, sender, tag).

```
if (rank == 0) {
    double vecteur[5] = { 1., 3., 5., 7., 22. };
    MPI_Send(vecteurs, 5, MPI_DOUBLE, 1, 101, commGlob);    }
else if (rank==1) {
    MPI_Status status;    double vecteurs[5];
    MPI_Recv(vecteurs, 5, MPI_DOUBLE, 0, 101, commGlob, &status);    }
```

Interlocking

Definition

- Interlocking is a situation where many processes are waiting for each other indefinitely to complete their messages.
- For example, process 1 waits to receive a message from process 0, and process 0 waits to receive a message from process 1.
- Alternatively, process 0 sends a message to process 1, and process 1 waits for a message from process 0 but with the wrong tag.
- Sometimes, identifying interlocking can be challenging.
- **Rule of thumb** : Be careful to ensure that each send has a corresponding receive with the correct tag and expeditor.

```
if (rank==0)
{
    MPI_Recv(rcvbuf, count, MPI_DOUBLE, 1, 101, commGlob, &status);
    MPI_Send(sndbuf, count, MPI_DOUBLE, 1, 102, commGlob);
}
else if (rank==1)
{
    MPI_Recv(rcvbuf, count, MPI_DOUBLE, 0, 102, commGlob, &status);
    MPI_Send(sndbuf, count, MPI_DOUBLE, 0, 101, commGlob);
}
```

Interlocking (more complicated cases)

```
MPI_Comm_rank(comm, &myRank ) ;
if (myRank == 0 )
{
    MPI_Ssend( sendbuf1, count, MPI_INT, 2, tag, comm);
    MPI_Recv( recvbuf1, count, MPI_INT, 2, tag, comm, &status);
}
else if ( myRank == 1 )
{
    MPI_Ssend( sendbuf2, count, MPI_INT, 2, tag, comm);
}
else if ( myRank == 2 )
{
    MPI_Recv( recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
              &status );
    MPI_Ssend( sendbuf2, count, MPI_INT, 0, tag, comm);
    MPI_Recv( recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
              &status );
}
```

Blocking and non blocking message

Definition

- Blocking message : Waits for the complete reception of the message before returning from the function.
- Non-blocking message : Posts the message to send or receive and returns from the function immediately.
- The status of a non-blocking message is updated in a request struct (not yet sent/received, sending/receiving, or sent/received).
- Allows testing or waiting for the message to be completed.

When to Use Non-blocking Messages ?

- When one can compute using other data during message exchanges to hide the message exchange cost.
- To simplify algorithms and ensure no interlocking situations occur.

Example using non blocking message

```
MPI_Request req;
if (rank == 0)
{
    double vecteur[5] = { 1., 3., 5., 7., 22. };
    MPI_Isend(vecteurs, 5, MPI_DOUBLE, 1, 101, commGlob, &req);
    // Some compute with other data can be executed here!
    MPI_Wait(req, MPI_STATUS_IGNORE);
}
else if (rank==1)
{
    MPI_Status status;    double vecteurs[5];
        MPI_Irecv(vecteurs, 5, MPI_DOUBLE, 0, 101, commGlob, &req);
    int flag = 0;
    do {
        // Do computation while message is not received on another data
        MPI_Test(&req, &flag, &status);
    } while(flag );
}
```

A scheme to avoid interlocking situations

The Scheme for All Processes

1. Perform receptions in non-blocking mode.
2. Perform sends in blocking mode (or non-blocking mode if you want to overlap message cost with computing).
3. Synchronize your receptions (waiting for completion or testing to overlap message cost with computing).

```
MPI_Request req; MPI_Status status;
if (rank==0)
{
    MPI_Irecv(rcvbuf, count, MPI_DOUBLE, 1, 101, commGlob, &req);
    MPI_Send(sndbuf, count, MPI_DOUBLE, 1, 102, commGlob);
    MPI_Wait(&req, &status);
}
else if (rank==1)
{
    MPI_Irecv(rcvbuf, count, MPI_DOUBLE, 0, 102, commGlob, &req);
    MPI_Send(sndbuf, count, MPI_DOUBLE, 0, 101, commGlob);
    MPI_Wait(&req, &status);
}
```

Buffered or Non-buffered Messages

Buffered Messages

- A non-blocking send is copied into a buffer before being sent.
- $\Rightarrow$ After calling a non-blocking send, the user can modify the sent data without changing the values to be sent.
- This is the default behavior when sending a small message.
- However, copying the data into a buffer incurs memory and CPU costs.
- Users can call send functions that don't copy the data into a buffer.
- It is the responsibility of the user to avoid changing data before the completion of the message !

Example of non buffered messages

```
std::vector<double> tab{3.,5.,7.,11.};
// Blocking non bufferized send
MPI_Ssend(tab.data(), tab.size(),
          MPI_DOUBLE, 1, 104, commGlob);
// OK, tab sent, can modify the buffer
tab[3] = 13.;
```

Right example

```
std::vector<double> tab{3.,5.,7.,11.};
// Non blocking non bufferized send
MPI_Issend(tab.data(), tab.size(),
    MPI_DOUBLE, 1, 104, commGlob,&request);
MPI_Wait(&request, &status);
// OK, tab sent, can modify the buffer
tab[3] = 13.;
```

Right example

```
std::vector<double> tab{3.,5.,7.,11.};
// Non blocking non bufferized send
MPI_Issend(tab.data(), tab.size(),
    MPI_DOUBLE, 1, 104, commGlob,&request);
// ERROR, we modify the buffer before
//        than the tab is sent !
tab[3] = 13.;
```

Wrong example

Collective Messages

What is Collective Communication

- Broadcast data from one process to all processes.
- Scatter data from one process to all processes.
- Gather data from all processes to one process.
- Reduce data (with arithmetic operation) from all processes to one/all processes.
- Scan data (with arithmetic operation) from all processes to all processes.
- All-to-all broadcast/scatter data.

Why Collective Communication

- Point-to-point communication is sufficient for many algorithms !
- However, for some parallel operations (broadcasting, reduction, scattering), the optimal algorithm depends on the network topology.
- Distributed parallel libraries provide collective communication that is optimized for various network topologies.
- The resulting algorithm is clearer.

Distributed Parallel Rules

- Ethernet data exchange is very slow compared to memory access : minimize data exchanges as much as possible.
- To hide data exchange costs, it's better to compute some values during the exchange of data : prefer using non-blocking message exchange.
- Each message has an initial cost : regroup data in a temporary buffer if needed.
- Ensure that all processes exit the program at the same time : try to balance the computing load among computing nodes.

Available Tools

- Program Ethernet layers (for specialists only !).
- Use dedicated libraries (MPI, PVM, ...).
- In all cases, data exchange must be explicit, done by calling functions provided by the library.
- It is better to design one's software with parallelism in mind from the beginning of the project.