RÉPUBLIQUE
FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

Programming parallel computers

Introduction

Xavier JUVIGNY, SN2A, DAAA, ONERA
xavier.juvigny@onera.fr

Course Parallel Programming
- September 20th 2023 -

[1] ONERA, [2] DAAA

Ce document est la propriété de l'ONERA. Il ne peut être communiqué à des tiers et/ou reproduit sans l'autorisation préalable écrite de l'ONERA, et son contenu ne peut être divulgué.
This document and the information contained herein is proprietary information of ONERA and shall not be disclosed or reproduced without the prior authorization of ONERA.

Table of contents

Overview

Parallel architecture

The main story

- Processors with multiples computing cores for faster computation
- Using simultaneously many cores for a unique application
- Performance benchmark in scientific computing given by the number of FLoating Operations Per Seconds (FLOPS)

Hardware implementation

- Many computing cores sharing a same main memory inside a computer
- Using many computers linked with a fast specialized ethernet connection
- Mixing both technologies above

Interests of parallel architecture ?

- **Gordon Moore's "Law"** : In 1965, Gordon Moore (one of Intel's fonder) observes that the number of transistors for each generation of processors doubles in eighteen months, doubling the computing power.
- In fact, it isn't a law, but it has been used by processors builder as a roadmap until 2000 years to raise the frequency of computing cores.
- **Limitations of Gordon Moore's Law** : The miniaturisation of transistors and the raising of their frequencies raises the heat inside the processor. Moreover, the miniaturisation is now at molecular scale and one must consider quantum effects (as tunnel effect) when making a processor.
- Nowadays, **Moore's law is still verified**, not by doubling the number of transistors inside a computing core, but rather by raising the number of computing cores inside a processor or a computer.

Parallel computing example (1)

Control command

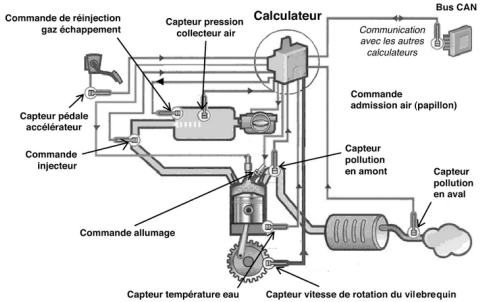


Figure – Car's control command

- Many tiny computers specialized for specifics tasks : ABS, motor optimization, lighting, air conditioning, wheel pressure optimization, mixing fuel/air, battery optimization, and so on.
- Computation must be terminated in constraint times.
- Lot of 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
- Lot of complex computations
- One computing core isn't enough to satisfy tiny real time constraints.
- **Solution** : Concurrent execution of interdependent tasks on many computing cores.

Parallel computation example (2)

Physical Simulation

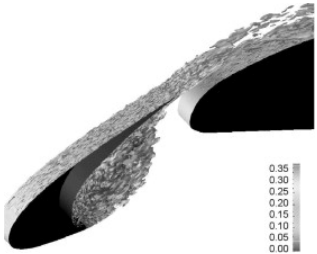


Figure – Turbulent noise generated by a plane's slate wing

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

Parallel computation example (3)

Artificial intelligence

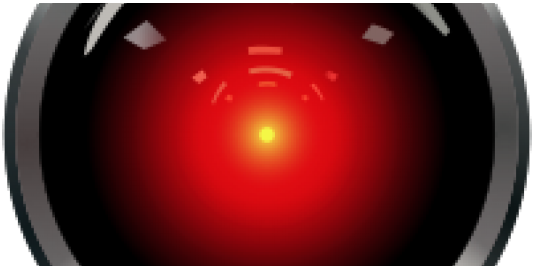


Figure – A very famous artificial intelligence (HAL)

- Deep learning used in AI to categorize pictures, automatic translations, cancerous cells detection, automatic vehicles, and so.
- In sequential, required more than one year to learn
- With GPGPU, required about some hours or few days
- March 2016 : Alphago wins versus world GO human champion (supervised learning).
- October 2017 : Alphago zero wins versus alphago at 100 games to zero (deep learning).

Parallel computation (4)

Picture treatment

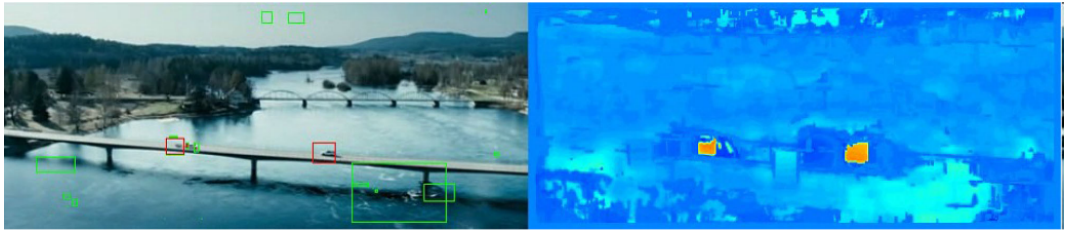


Figure – Real time constraint treatment of a video with 30 frames/s (resolution 1920×1080 pixels)

- Needed for optical captors for navigation of autonomous vehicles, for super-resolution picture issued from low resolution video, and so on.
- Lot of computations needed (PDE equation to solve).
- Must use GPGPU and parallel algorithms to have real time constraint.

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

Shared memory architecture

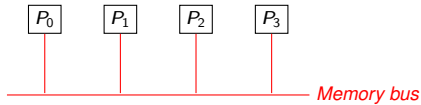


Figure – Simplified scheme of shared memory parallel architecture

Many computing cores share the same main memory.

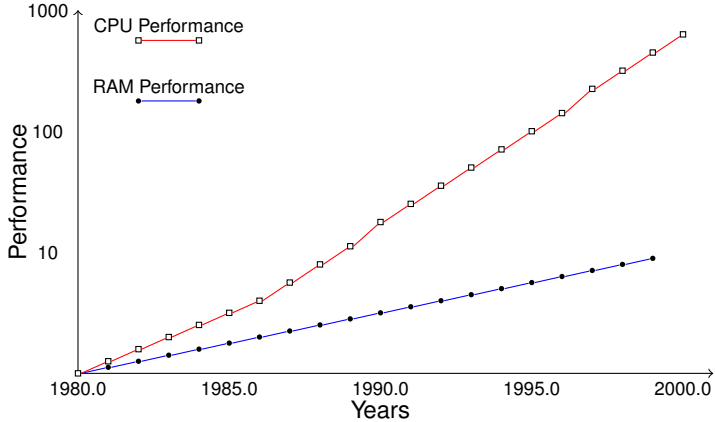
Examples

- Recent multi-cores processors
- Graphic cards with 3D acceleration
- Phones, pad, etc.

Memory access problem

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

Memory access



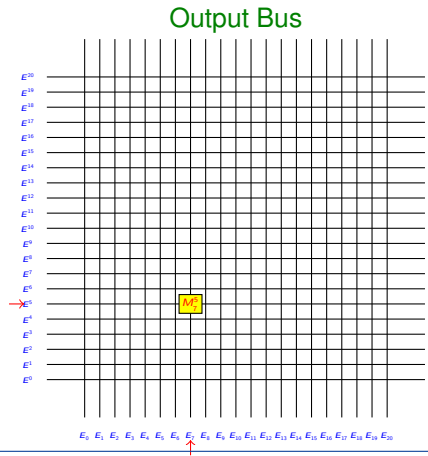
Latency memory example (Haswell architecture)

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

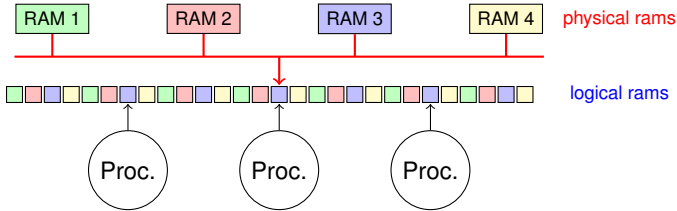
Conclusion

- Memory is slower and slower comparing to the instruction execution of the processor.
- It's even worse with multicore architecture !

How does RAM work ?



Interleaved RAMs

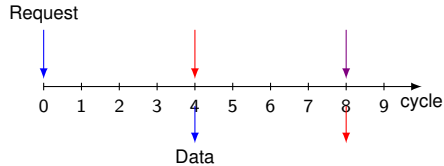


Interleaved memory

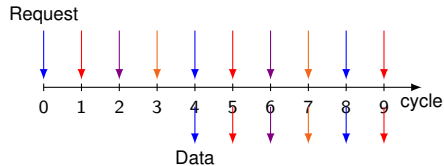
- Many physical memory units interleaved by the memory bus
- Number of physical memory units $\equiv$ number of ways
- Number of contiguous bytes in a unique physical memory $\equiv$ width of way
- Quadratic cost in € to build, relative to number of memory units !

Interleaved memory access

Classic memory access



Four ways interleaved memory access



Cache memory

Consequences of grid architecture of RAMS

Bigger is a memory, bigger is its grid, slower is the read and write access.

Cache memory

- Fast small memory unit where one stores temporary data
- When multiple access to a same variable in a short time, speedup the read or write access
- Cache memory managed by the CPU (but cache memory for GPU can be managed by the programmer)
- **Consequence** : to optimize his program, the programmer must know the strategies used by the CPU.

Cache memory

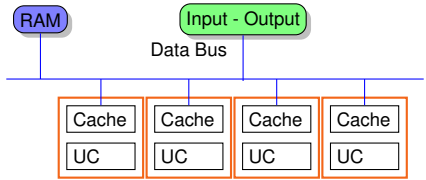
CPU Strategy

- **Cache line** : store contiguous memory variables in cache (64 bytes on Intel processor)
- **Associative memory cache** : each cache memory address mapped on fixed RAM address (with a modulo)

Consequences

- Better to have contiguous access in memory
- Better to use as soon as possible data stored in cache
- Spatial and time localization of data

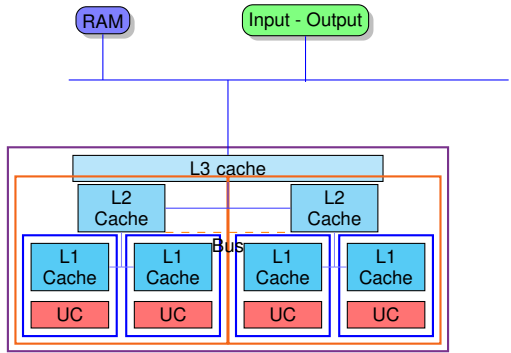
Memory organization on multi-processor computer



Data coherence between memory caches :

- A unique cache contains the datum : value is valid, no synchronization needed.
- Datum shared with another memory cache : at each access, verify if the datum is modified by another core and write as invalid when modifying its value.
- Modify the value in the cache : value now not valid in RAM, update the value in RAM if another core reads the value.
- Value is invalid for cache : the next read of this value must access the value in RAM.

Many cores cache organization



Same issue with cache consistency, but need coherence of data between cache levels. Complexity raises with the number of cache levels.

Tools for shared memory computation

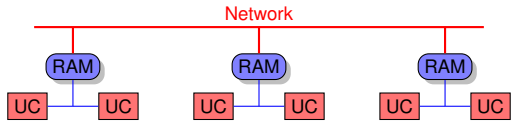
Many tools can be used to use many "threads" and the synchronization in memory. The most used are :

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

But the programmer must take care of memory conflict access :

- when a thread writes a datum and some other threads read simultaneously that same datum ;
- when some thread writes at the same datum ;
- doesn't rely on the instruction order in the program (out-of-order optimization by compiler and processor) !

Distributed memory



- Each computing unit can read/write on local RAM : the set containing the computing unit and the RAM is called **Computing node**
- The data are exchanged between computing nodes through a specialized bus or specific ethernet link
- On ethernet link, it's the responsibility of the programmer to exchange explicitly the data between computing nodes
- Need specific efficient algorithms and libraries
- Possible to compute on many thousands of computing cores
- Only limited by electricity consumption (linear cost)

Distributed parallelism context

All libraries managing the distributed parallelism computation provide similar functionalities.

Running a distributed parallel application

- An application is provided to the user to run his application(s) on a wanted number `nbp` of computing nodes (given when running the application)
- The computing nodes where the application(s) is launched is defined by default or in a file provided by the user
- The default output for all processes are the terminal output from which was launched the application
- A communicator (defining a set of processes) is defined by default including all launched processes (`MPI_COMM_WORLD`)
- The application gives a unique number for each process in a communicator (numbering from 0 to `nbp`-1)
- All processes terminate the program at the same time

Managing the context in your program

- Call initialization of parallel context before using other function in the library (`MPI_Init`)
- Get the number of processes contained by the communicator (`MPI_Comm_size`)
- Read the rank of the process inside the communicator (`MPI_Comm_rank`)
- After calling the last library function, call the termination of parallel context to synchronize processes (`MPI_Finalize`, if not done, crash your program)

```
#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

A process sends some data in a message to another process which receives this message.

Constitution of a data message to send

- The communicator used to send the data
- The memory address of the contiguous data to send
- The number of data to send
- The type of the data (integer, real, user def type, and so.)
- The rank of 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 where store the received data
- The number of data to receive
- The type of the data (integer, real, user def type, and so.)
- 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 each other for an infinite time to complete their messages
- For example, process 1 waits to receive a message from 0 and 0 waits to receive a message from 1
- Or process 0 sends a message to 1 and process 1 waits a message from 0 but with wrong tag !
- Sometimes, interlocking can be very hard to find !
- **Rule of thumb** : be careful that each send has a corresponding receive, with 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 : wait the complete reception of the message before returning from the function
- Non blocking message : post the message to send or receive and return from the function immediatly !
- The status of non blocking message is updated in a request struct. (not yet sent/recv, sending/receiving or sent/received)
- Allows to test or wait for the message to be completed

When to use non blocking message ?

- When one can compute using other data during messages exchanges to hide message exchange cost
- To simplify algorithms and to 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. Do receptions in non blocking mode
2. Do send in blocking mode (or non blocking mode if you want to overlay message cost with computing)
3. Synchronize yours receptions (waiting for completion or testing to overlay 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 in a buffer before to be sent
- $\Rightarrow$ After calling a non blocking send, the user can modify the sent data without changing the values to send
- It's the default behavior when we send a small message
- But the copy of the data in a buffer has a memory and CPU cost !
- We can call send functions which doesn't copy the data in a buffer
- It's the responsibility of the user to avoid to change 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 all algorithms !
- But, for some parallel operations (broadcasting, reduction, scattering), the optimal algorithm depends on net topology
- Distributed parallel libraries provide collective communication which are optimized for all net topology
- The resulting algorithm is clearer

Distributed parallel rules

- Ethernet data exchange is very slow compared to memory access : limit as possible the data exchanges
- To hide data exchange cost, it's better to compute some values during the exchange of data : prefer to use non blocking message exchange
- Each message has an initial cost : prefer to regroup data in a temporary buffer if needed
- All processes quit 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 library (MPI, PVM, ...)
- In all cases, data exchange must be explicit, done by calling functions provided by the library
- Better to think one's software in parallel at the beginning of the project