CEMRACS - Intro MPI-OpenMP

D. Lecas

23 juillet 2012

Sommaire I

1 Introduction 4

2 MPI and distributed memory 17

3 OpenMP and shared memory 20

4 Introduction to code optimization 24

Before

IDRIS course materials

This slides are extract from IDRIS tutorials : `https://cours.idris.fr/`

- MPI tutorial(J. Chergui, I. Dupays, D. Girou, P.-F. Lavallée, D. Lecas, P. Wautelet)
- OpenMP tutorial(J. Chergui, P.-F. Lavallée)
- Hybrid tutorial (P.-F. Lavallée, P. Wautelet)

1 – Introduction

1.1 – Moore's Law and Electric Comsumption

Statement

Moore's law says that the number of transistors that can be placed inexpensively on an integrated circuit doubles every two years.

Electric consumption

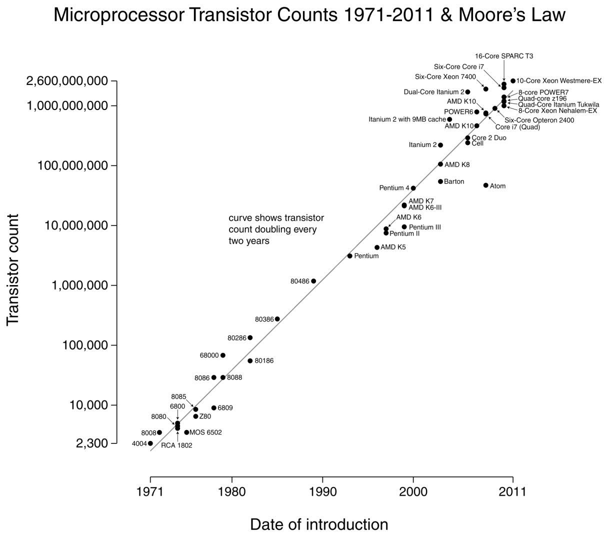
- Dissipated electric power = $frequency^3$ (for a given technology).
- Dissipated power per cm^2 is limited by cooling.
- Energy cost.

Moore's Law and Electric Consumption

- The frequency of processors does not increase any more due to the prohibitive electric consumption (maximum frequency is blocked around 3 GHz since 2002-2004).
- The number of transistors per chip continues to double every two years.

=> the number of cores per chip increases (the Intel Nehalem have up to 10 cores and support 20 threads simultaneously ; the AMD Interlagos have 16 cores)

1 – Introduction
1.1 – Moore's Law and Electric Comsumption



CC BY-SA 3.0, `http://en.wikipedia.org/wiki/Moore%27s_law`

1 – Introduction

1.2 – The Memory Wall

Causes

- The throughput towards the memory increases less than the computing power of processors.
- The latencies (access time) of the memory decrease very slowly.
- The number of cores per memory module increases.

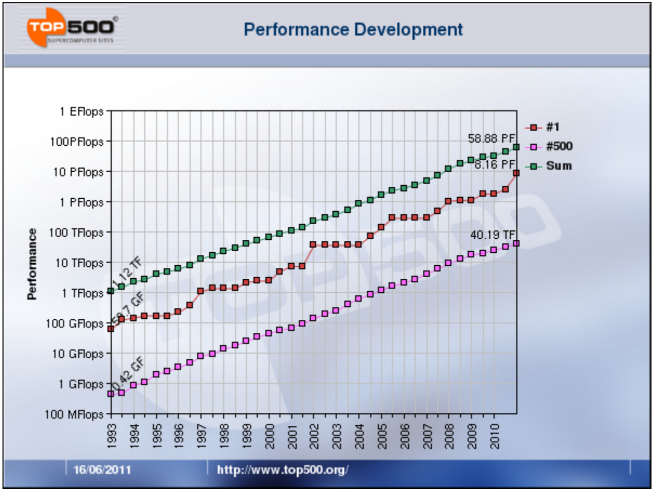
Consequences

- The gap between the theoretical performances of the cores and the memory increases.
- The processors spend more and more cycles waiting for data.
- It is increasingly difficult to exploit the performance of processors.

Partial Solutions

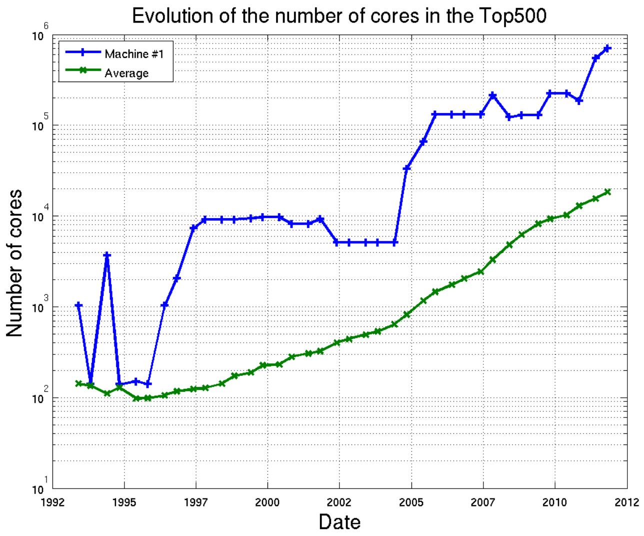
- The addition of cache memories is essential.
- Parallelization of accesses via many memory banks like on the vector architectures (Intel Sandy Bridge : 4 channels and AMD Interlagos : 4).
- If the frequency of the cores stagnates or falls, the gap could be reduced.

1 – Introduction
1.3 – As for Supercomputers



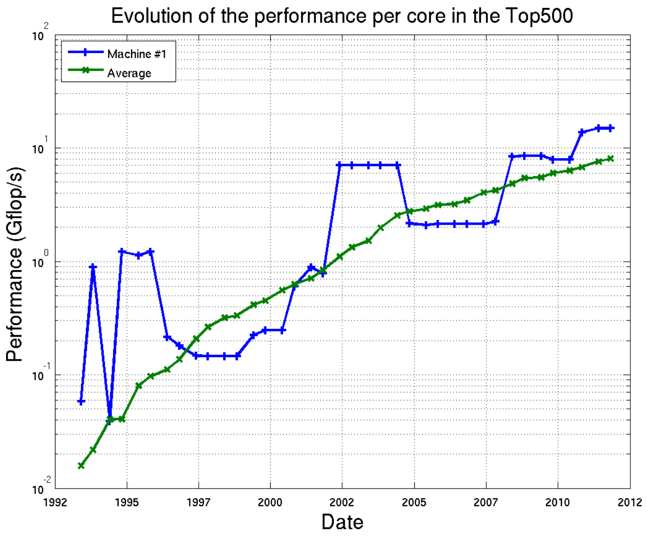
1 – Introduction

1.3 – As for Supercomputers



1 – Introduction

1.3 – As for Supercomputers



1 – Introduction
1.3 – As for Supercomputers

Technical Evolution

- The computing power of supercomputers doubles every year (faster than Moore's law, but electric consumption also increases).
- The number of cores increases rapidly (massively parallel and many-cores architectures).
- Emergence of hybrid architectures (GPU, PowerXCell 8i or FPGA and standard processors for example).
- The architecture of the machines becomes more complex and the number of levels increases (processors/cores, access to the memory, network and I/O).
- The memory by core stagnates and starts to decrease.
- The performance by core stagnates and is on some machines much more lower than on a simple laptop (IBM Blue Gene).
- The throughput towards the disks increases more slowly than the computing power.

1 – Introduction

1.4 – Amdahl's Law

Statement

Amdahl's law predicts the theoretical maximum speedup obtained by parallelizing ideally a code, for a given problem and a fixed problem size.

$$Sp(P) = \frac{T_s}{T_{//}(P)} = \frac{1}{\alpha + \frac{(1-\alpha)}{P}} < \frac{1}{\alpha} \quad (P \to \infty)$$

with Sp the speedup, T_s the execution time of the sequential code (monoprocessor), $T_{//}(P)$ the execution time of the code ideally parallelized on P cores and α the non parallelizable part of the application.

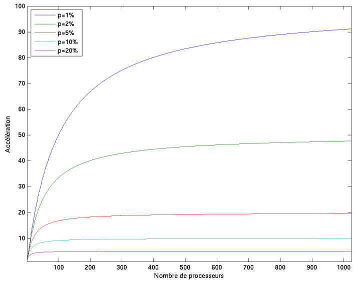
Interpretation

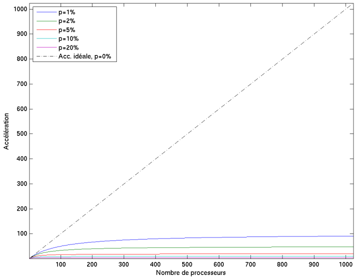
Regardless of the number of cores, the speedup is always less than the inverse of the percentage that the purely sequential fraction represents.

Example : if the purely sequential fraction of a code represents 20% of the execution time of the sequential code, then regardless of the number of cores, we will have : $Sp < \frac{1}{20\%} = 5$

Theoretical Maximum Speedup

Cores	α (%)								
	0	0.01	0.1	1	2	5	10	25	50
10	10	9.99	9.91	9.17	8.47	6.90	5.26	3.08	1.82
100	100	99.0	91.0	50.2	33.6	16.8	9.17	3.88	1.98
1000	1000	909	500	91	47.7	19.6	9.91	3.99	1.998
10000	10000	5000	909	99.0	49.8	19.96	9.99	3.99	2
100000	100000	9091	990	99.9	49.9	19.99	10	4	2
∞	∞	10000	1000	100	50	20	10	4	2





1 – Introduction
1.5 – Gustafson-Barsis' Law

Statement

Gustafson-Barsis' law predicts the theoretical maximum speedup obtained by parallelizing ideally a code, for a problem of constant size by core and by supposing that the execution time of the sequential fraction does not increase with the overall problem size.

$$Sp\,(P) = P - \alpha\,(P - 1)$$

with Sp the speedup, P the number of cores and α the non-parallelizable part of the application.

Interpretation

This law is more optimistic than Amdahl's law because it shows that the theoretical speedup increases with the size of the studied problem.

1 – Introduction
1.6 – Consequences for users

Consequences for the applications

- It is necessary to exploit a large number of relatively slow cores.
- The memory by core tends to decrease, necessity to carefully manage it.
- A level of parallelism always more important is needed to use the modern architectures (computing power, memory size).
- The IO becomes at the same time an increasing problem.

Consequences for the developers

- End of days when it sufficed to wait to gain in performance (stagnation of the computing power per core).
- An increased necessity to understand the hardware architecture.
- It becomes increasingly complicated to develop on its own (need of experts in HPC and multi-disciplinary teams).

Evolution of Programming Methods

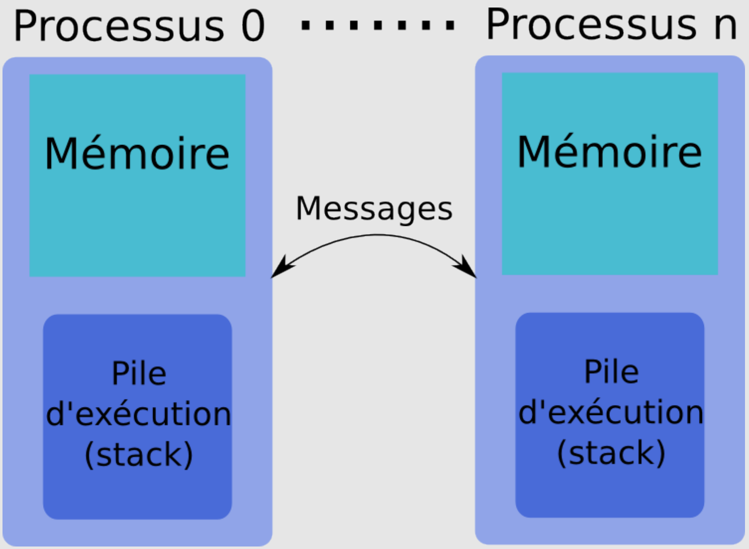
- MPI is still predominant and it will remain so for some time (a very important community of users and a majority of current applications).
- The MPI-OpenMP hybrid approach grows and seems to be the preferred one for supercomputers.
- GPU programming grows, but it is still very immature.
- Other forms of hybrid programming are also tested (MPI + GPU...) with MPI as common ingredient.
- New parallel programming languages appear (UPC, Coarray- Fortran, PGAS languages, X10, Chapel...), but they are in experimental phase (at very variable levels of maturity). Some of these languages are very promising. It remains to be seen whether they are going to be used in real applications.

2 – MPI and distributed memory

Presentation of MPI

- Parallelization paradigm for distributed memory architecture based on the use of a portable library.
- MPI is based on an approach of communications between processes by message passing.
- MPI provides different types of communications :
    - Point-to-point
    - Collective
    - One-sided
- MPI provides also the following functionalities (non-exhaustive list) :
    - Execution environment
    - Derived datatypes
    - Communicators and topologies
    - Dynamic process management
    - Parallel I/O
    - Profiling interface

Distributed memory



2 – MPI and distributed memory

Limitations of MPI

- The final speedup is limited by the purely sequential fraction of the code (Amdhal's law).
- The additional costs related to the MPI library and the management of the load balancing limit the scalability.
- Some types of collective communications become more and more time-consuming when the number of processes increases (for example `MPI_Alltoall`).
- There is no distinction between processes running in shared or distributed memory, but the impact on the communications performance is crucial. Most implementations take this into account, but the standard does not allow any feedback towards the application.
- There are fewer means included in the standard in order to match the hardware and the MPI processes (process mapping for example). It can often be done out of the MPI standard.

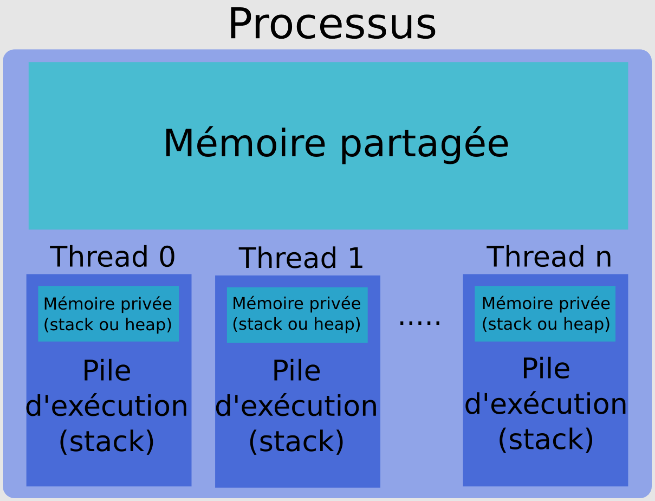
3 – OpenMP and shared memory

Presentation of OpenMP

- Parallelization paradigm for shared-memory architecture based on directives to be inserted in the (C, C++, Fortran) source code.
- OpenMP consists of a set of directives, a library of functions and a set of environment variables.
- OpenMP is an integral part of any recent Fortran/C/C++ compiler.
- It can manage :
  - the creation of threads,
  - the work-sharing between these threads,
  - the (explicit or implicit) synchronizations between all the threads,
  - the status of variables (private or shared).

3 – OpenMP and shared memory

Schematic Drawing



3 – OpenMP and shared memory

3.1 – Limitations of OpenMP

Limitations of OpenMP (1)

- The scalability of a parallel code is physically limited by the size of the shared-memory node on which it operates.
- In practice, the cumulated memory bandwidth inside an SMP node can further limit the number of cores used efficiently. The contentions due to the memory bandwidth limitation can often be removed by optimizing the use of caches.
- Pay attention to the implicit additional costs of OpenMP during the creation of threads, from the synchronization between the (implicit or explicit) threads or from the work-sharing as the processing of parallel loops for example.
- Other additional costs directly linked to the target machine architecture can also affect the performances, like false sharing on shared caches or the NUMA aspect of the memory accesses.
- The binding of threads on physical cores of the machine is the responsibility of the system (accessible to the developer by using some libraries). Its impact on the performances can be very important.

3 – OpenMP and shared memory

3.1 – Limitations of OpenMP

Limitations of OpenMP (2)

- OpenMP does not manage the data locality aspect, which poses a real problem on NUMA-based architecture and does not allow any use of GPGPU-based architecture.
- OpenMP is not suitable for dynamic problems (i.e. whose work load evolves rapidly during the execution).
- As for any code parallelization, the final speedup will be limited by the purely sequential code fraction (Amdahl's law).

4 – Introduction to code optimization

Definition

- Optimizing a computer code consists of reducing its needs of resources.
- These latter ones are varied but in general we speak of elapsed time.
- The memory or disk-space consumption falls also in this category.

4 – Introduction to code optimization

Why Optimizing ?

Optimizing an application can bring a certain number of advantages :

- Faster obtaining of results by a reduction of elapsed time ;
- Possibility of obtaining more results with the attributed hours ;
- Possibility of making larger computations ;
- Better understanding of the code, the machine architecture and their interactions ;
- Competitive advantage in comparison to other teams ;
- Detection and correction of bugs through the re-reading of code sources.

The application performance enhancement also provides :

- A reduction of the energy consumption of computations ;
- A better machine use (the cost of buying, maintenance and use of a supercomputer is not negligible). At IDRIS, each attributed hour represents an expense for all the scientific community ;
- Freeing-up of resources for other research groups.

4 – Introduction to code optimization

Why not Optimizing ?

- Lack of resources or means to do so (lack of staff, time,skills...) ;
- Decrease in the code portability (so many optimizations are specific to the machine architecture) ;
- Risk of performance loss on other machines ;
- Decrease in the readability of sources and more difficult maintenance ;
- Risk of inadvertently introducing bugs ;
- Unsustainable code ;
- Sufficiently fast code (it is useless to optimize a code which already provides results in a reasonable time) ;
- Already optimized code.

4 – Introduction to code optimization

When to Optimize ?

- An application should not be optimized before it works correctly, because of
  - risks of inadvertently introducing new bugs,
  - decrease in readability and code understanding and
  - risks of optimization of procedures that will be abandoned or that will not be used or will be used very little or will be totally rewritten.
- Do not launch into optimization unless the application is very slow or does not allow making large computations in a reasonable time.
- If you are fully satisfied with the performances of your application, the investment might not be necessary.
- Have some time ahead.

4 – Introduction to code optimization

How to Optimize ?

- Above all : make a sequential and parallel profiling with a realistic test set to identify the critical zones
- Optimize where the resources are most consumed
- Check every optimization : are the results always correct ? Are the performances really improved ?
- Question : if the gain is little, is it necessary to keep the optimization ?

What to Optimize ?

- The sequential performances
- The MPI communications, the OpenMP performances and the scalability
- The I/O

4 – Introduction to code optimization

Sequential Optimization

- Algorithms and conception
- Libraries
- Compiler
- Caches
- Specialized processing units (SIMD, SSE, Altivec...)
- Others

4 – Introduction to code optimization

Optimization of the MPI Communications, OpenMP and the Scalability

- Algorithms and conception
- Load balancing
- Computation-communication overlap
- Process mapping
- Hybrid programming
- Others

4 – Introduction to code optimization

Optimization of I/O

- Read and write only the necessary
- Reduce the precision (simple precision instead of double)
- Parallel I/O
- Libraries (MPI-I/O, HDF5, NetCDF, Parallel-NetCDF...)
- Hints MPI-I/O
- Others

4 – Introduction to code optimization

4.1 – Gprof

Principle

- learn where your program spent its time ;
- which functions called which other functions.

Utilization

- Compile with -pg (compilation and liking)
- Execute the executable (add an overhead) on output a file gmon.out
- Print the profil with gprof executable gmon.out

4 – Introduction to code optimization

4.1 – Gprof

Profil

```
 %   cumulative self              self   total
time   seconds seconds  calls  ms/call ms/call  name
63.3   339.88  339.88  5000000  0.07    0.07  hy_ut_NMOD_riemann [4]
 8.0   382.82   42.94  5000000  0.01    0.01  hy_ut_NMOD_trace [6]
 7.9   425.07   42.25    20000  2.11   25.89  hy_pr_NMOD_godunov [3]
 7.8   466.93   41.86  5000000  0.01    0.01  hy_ut_NMOD_constoprim [5]
```

Call graph

```
                              called/total    parents
index %time   self    descen  called+self     name              index
                              called/total    children
              42.25   475.52   20000/20000    hy_main [1]
[3]    96.5   42.25   475.52   20000          hy_pr_NMOD_godunov [3]
             339.88     0.00 5000000/5000000 hy_ut_NMOD_riemann [4]
              41.86    22.62 5000000/5000000 hy_ut_NMOD_constoprim [5]
              42.94    16.38 5000000/5000000 hy_ut_NMOD_trace [6]
              11.38     0.00 5000000/5000000 hy_ut_NMOD_cmpflx [10]
               0.24     0.00   20000/20000    hy_ut_NMOD_make_boundary [13]
```

4 – Introduction to code optimization

4.2 – Cache

Memory and cycles

Memory	*Register*	*L1*	*L2*	*RAM*
Cycles	≤ 1	≈ 3	≈ 14	≈ 240

Use the cache as much as possible. Valgrind with cachegrind module can be usefull to see cache use.

Don't follow this Example

```
! Bad cache use
do i=1,n
  do j=1,n
    a(i,j) =
  enddo
enddo
```