[]

Agenda

HPC and modeling

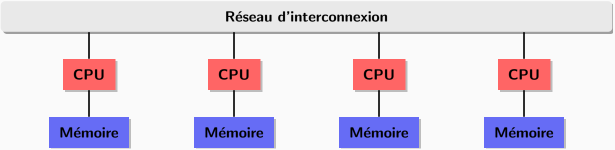
Chapter 3 – Models and Patterns (MPI)

M2 – MSIAM

November 14, 2017

Agenda

Distributed memory



Why?

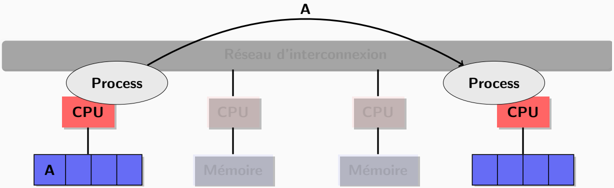
- OpenMP targets shared memory architectures, but the number of processing units is limited.
- On the different architectures with distributed memory, the main bottleneck is the communication bus between the various components.
- We also wish to use the different computing units available on a network.
- Communication protocols on a network are powerful but are too complex and fastidious to be use for massive communication between processors.
- In order to simplify the communications protocols in the case of scientific computing a common library is needed.
- MPI standard allows to manage heterogeneous systems: clusters of PC, playstation or high performances systems with millions of cores.
- It allows hybrid programming for different systems.

The developer still need to handle some communications and how the data are shared between the various resources.

Message Passing Communication

- A program execute several processes simultaneously. Some small part of the code may differ between the processes.
- Each process own is data.
- Data from other processes cannot be read directly.
- Data that are stored locally are exchanged between the processes using specialised communication protocols.

Exchange of message



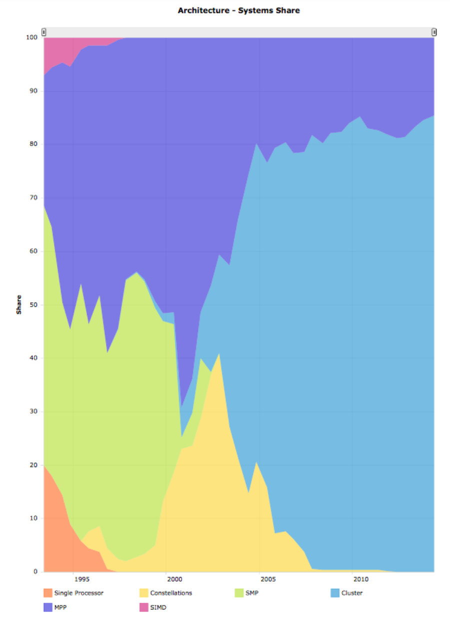
MPI standard

- MPI : message passing interface
- Unify the different implementations of the hardware manufacturers involve in high perfomances computing.
- The first draft of the standard was proposed at Supercomputing 1993.
- Standard practical, portable, efficient and flexible
  - can be used in C, C++, Fortran
  - avoid memory transfer and allows communications and computations simultaneously.
  - supported by a large number of manufacturers.
  - interface close to already existing protocols (PVM,...)
  - independence of the semantic with respect to the programming language.
  - thread-safe

Some short history

- November 92 : creation of a working group.
- November 93 : presentation of the draft.
- 1995–2008 : Publication and improvements of MPI 1.1 -> 1.3
- 1997–2008 : Publication and improvements of MPI 2.1 -> 2.2
- 2008 : Fusion of MPI 1.3 and MPI 2.0
- 2007 : Creation of a new working group.
- 2012 : MPI 3.0 with support for many-core architectures.

Supercomputing



Message passing

- Relies on exchanging message between processes to transfer data, synchronization of processes and global operations.
- MPI provides a complete infrastructure for managing the communications..
- Relies heavily on single program multiple data.
- Each process has its own data without being able to access others.
- The sharing of data is left to the programmer.
- Exchanges are done within a global space: a communicator.
- Each process is identified by a rank (int) within the communicator or sub-communicator.

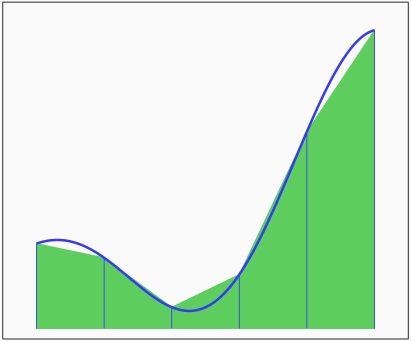
Organisation of MPI

1. environment
2. point to point communications
3. global communications
4. derived types
5. communicators
6. I/O

Compiling – Executing

- Include the file`"mpi.h"`
- Collection of wrapper for `gcc`: compile everything with `mpicc`, `mpic++` or `mpif90`
- Wrappers allows to use the compiler with the good options.
- To run an MPI program, we should specify the communication protocol between the processes: `export RSHCOMMAND=ssh`
- With the library OpenMPI
  `mpirun -machinefile file -np X executable [options]`
- `X` is the number of cores to use.
- `file` specify the list of processors and how to use them.
  ```
  host1.exemple.com [slots=X1 max\_slots=Y1]
  host2.exemple.com [slots=X2 max\_slots=Y2]
  ```
  with
  `X` number of core/CPU on the compute node.
  `Y` max number of processes that MPI can use on this node.

Example 1: Integral Computation



Trapezium formula is given by $\forall f \in C^2([a;b])$, $\exists \xi \in [a;b]$

$$I = \frac{h}{2}\left(f(a) + 2\sum_{k=1}^{n-1} f(a+kh) + f(b)\right) - (b-a)\frac{h^2}{12} f^{(2)}(\xi)$$

Example 1: Pseudo-code

```
Input b, a, n
h = (b-a)/n
I = (f(a) + f(b))/2.0;
for (i = 0; i<= n-1; i++)
{
  x_i = x_i + h
  I += f(x_i)
}
I = h*I
```

Example 1: Parallel Pseudo-code

```
Find b, a, n
h = (b-a)/n
local_n = n/n_p
local_a = a + id * local_n*h
local_b = local_a + local_n*h
local_I = Trap(local_a, local_b, local_n)

If (id == 0)
{
  I = local_I;
  for (i = 1; i<= n_p; i++)
  {
    I += local_I;
  }
  Display I;
}
```

Example 2: Matrix Multiply

Let $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$. Then $C = A \times B$, $C \in \mathbb{R}^{m \times p}$ is defined by

$$\forall 1 \leq i \leq n, 1 \leq j \leq p, c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Example 2: Pseudo-code

```
Input A, B, n, m, p
for(i = 1; i<= n;i++)
{
  for(j = 1; j<= p;j++)
  {
    for(k = 1; k<= m;k++)
    {
      C[i][j]= C[i][j]+ A[i][k] x B[k][j]
    }
  }
}
```

Example 2: Parallel Pseudo-code

```
Input A, B, n, m, p
l_n = n/nn_p
for(i = id*l_n +1; i<= (id+1)*l_n;i++)
{
  for(j = 1; j<= p;j++)
  {
    for(k = 1; k<= m;k++)
    {
      C[i][j]= C[i][j]+ A[i][k] x B[k][j]
    }
  }
}
```

Example 3: Gaussian Elimination – Pseudo-code

```
for k = 1 ... m:
  Find pivot for column k:
  i_max  := argmax (i = k ... m, abs(A[i, k]))
  if A[i_max, k] = 0
    error "Matrix is singular}"
  swap rows(k, i_max)
  Do for all rows below pivot:
  for i = k + 1 ... m:
    Do for all remaining elements in current row:
    for j = k ... n:
      A[i, j]  := A[i, j] - A[k, j] * (A[i, k] / A[k, k])
    Fill lower triangular matrix with zeros:
    A[i, k]  := 0
```

Agenda

Choice criterion

- Three parameters may influence the choice of a kind of parallelism
  - Flexibility: support of different programming constraints. Should adapt to different architectures.
  - Efficiency: better scalability.
  - Simplicity: allows to solve complex problem but with a low maintenance cost.
- For each model of parallelism, we will expose the strengths and weaknesses for each elements.

Dependency Design Space

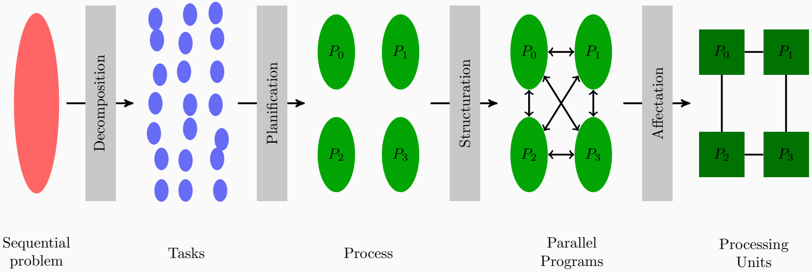
- The analysis is made on three elements
  - Grouping the data

    time dependency
    collection of data
    independence
  - Scheduling

    Identify which data are requires for executing a specific task.
    Identify the tasks creating the different data.
  - Sharing the data

    Identify the data shared between tasks.
    Manage access to data.

Foster design



Decomposition: idea

- Identify the elements that allows parallel processing and determine the granularity of the decomposition.
- Break up computation into tasks to be divided among processes
  - tasks may become available dynamically.
  - number of tasks may vary with time.
- Enough tasks to keep processors busy: the number of tasks available at a given time is an upper bound on achievable speedup.

Decomposition: focus

How to decompose the code in order to achieve maximum parallelism?

- Focus on data: domain decomposition
  partition data first into elementary blocks of independent data then associate computation tasks with data.
- Focus on computation: functional decomposition
  partition computation first then associate data to tasks.
- Often, we use a combination of this decomposition.

Domain decomposition

- Divide data into pieces of approximately equal size: data granularity.
- Partition computation by associating each operation with the data on which it operates.
- Set of tasks = (data, operations)

Use case: problems with large central data structures.

Example: manipulation of 3D data on a grid.

Functional decomposition

- Determine set of disjoint tasks.
- Determine data requirements of each task.
- If requirements overlap, communication is required.

Use case: problems without central data structures or to different parts of a problem.

Decomposition: Checklist

1. The granularity is controlled by the number of available processing units. The more tasks the better.

   ➜ improves flexibility in the design.

2. Limit the number of redundancy in data and computations.

   ➜ improves scalability for large problem.

3. Tasks should be of similar sizes.

   ➜ improves load balancing.

4. Number of tasks should depend on the size of the problem.

   ➜ improves efficiency.

5. All decomposition should be considered.

   ➜ check for flexibility.

Communication

Describe the flow of information between the tasks.

- Structure: relation between producers and consumers.
- Content: volume of data to exchange.

We should

- Limit the number of communication operations.
- Distribute communications among tasks.
- Organize communication in such a way that they are concurrent to operations.

Conceptual structure of a parallel program.

Communication: structure

Strong impact of decomposition on communication requirements

- Functional decomposition: data flow between the tasks.
- Domain decomposition: volume of data to perform a computation can be challenging or requires input from several other tasks.

Communication: types

- Local or global: small set of tasks or all?
- Structured or unstructured: grid or graphs?
- Static vs dynamic: known at the start of the program?
- Synchronous or asynchronous: cooperatives tasks?

Communication: Checklist

1. Load balancing of the communication operations.
   ➜ improves scalability.
2. Small communication pattern.
   ➜ improves scalability.
3. Communication are concurrent to computations.
   ➜ improves scalability.
4. Computations are concurrent to communications in different tasks
   ➜ improves scalability.

Agglomeration

After partitioning and communication steps, we have a large number of tasks and a large amount of communication. Need to combine into large blocks

- Increase granularity: reduce communication costs.
- Maintain flexibility: improve scalability.
- Reduce engineering costs: increase development overhead.

Agglomeration: Checklist

1. Communication costs are reduced.
2. Replication of data preserved scalability.
3. Replication of computation preserved performances.
4. Tasks are load balanced in term of computation and communication.
5. Scalability is preserved.

Mapping

Where to execute each tasks?

- Tasks that executes concurrently are placed on different processing units: increase concurrency.
- Tasks that communicates frequently are placed on the same processing units: increase locality.

Mapping is NP-complete

Mapping: strategies

- Static mapping: equal-sized tasks, structured communication.
- Load balancing: variable amount of work per tasks or unstructured communication.
- Dynamic load balancing: variable number of computation and communication per task.
- Task scheduling: short tasks.

Mapping: Checklist

1. SPMD algorithm: consider dynamic task creation.
   ➜ Simpler algorithm.
2. Dynamic task creation: consider SPMD algorithm.
   ➜ Greater control over scheduling of computation and communication.
3. Centralized load-balancing: verify manager does not become bottleneck.
4. Dynamic load-balancing: consider probabilistic/cyclic mappings.
5. Probabilistic/cyclic methods: verify that number of tasks is large enough.

Next time...

Lab on distributed computing