HPC and modeling

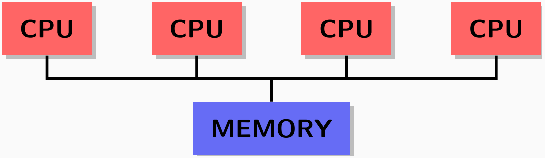
Chapter 1 – A first take on parallelism (2)

M2 – MSIAM
October 17, 2018

Hardware

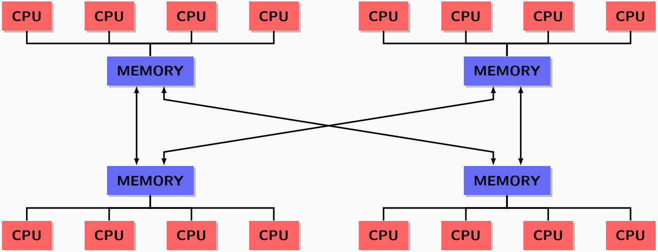
Shared memory (1)

- All the processors shares the same memory space. They communicate using reading and writing shared variables.
- Each processing unit carry out its task independently but modification of shared variables are instantaneous.
- Two kind of shared memories
  - SMP (Symmetric MultiProcessor) – All the processors share a link to the memory. Access to the memory is uniform.



Shared memory (2)

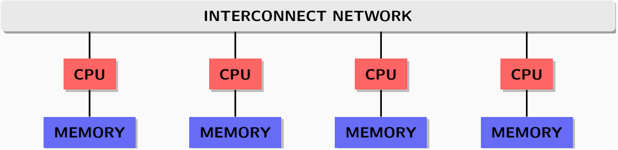
- NUMA (NonUniform Memory Access) – All the processors can access to the memory but not uniformly. Each processor has a preferred access to some memory part.



  - Decrease the risk of bottleneck to memory access.
  - Local memory cache on each processor to mitigate the effect of non-uniform access.

Distributed memory

- Each processor has its own memory. There is no global memory space.
- Each processor communicate with the others using messages.
  - Modification of variables are local and only the processor managing the memory can access it.
  - Each processor work independently on its own set of variables.
  - The speed of the resolution depends on the architecture: network, topology, processors.
  - Can scale easily.



Software

Program execution abstraction

Process

Usually, multiple processes, each with their own associated set of resources (memory, file descriptors, etc.), can coexist

Thread

- Typically smaller than processes
- Often, multiple threads per one process
- Threads of the same process can share resources

Task

- Typically smaller than threads
- Often, multiple tasks per one thread
- In parallel programming context: user-level construct

Parallel programming models

Classification according to process interaction

Message passing

- Parallel processes exchange data by passing messages
- Examples: PVM, MPI

Shared memory

- Parallel threads share a global address space
- Examples: POSIX threads, OpenMP

Implicit

- Process interaction is not visible to the programmer
- Examples: HPF

Parallel programming models

Classification according to problem decomposition

Data parallelism

- Independently process different parts of the problem data (e.g. an array of numbers)
- Maps well to SIMD

Task parallelism

- Independently execute different workpackages
- Might be heterogeneous, communicating packages

Implicit

- Programmer does not explicitly decompose problem

Foster methodology

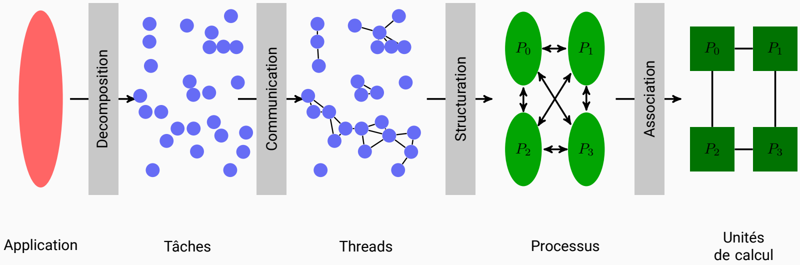
Choice criterion

- 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

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 decompositions.

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

- The granularity is controlled by the number of available processing units. The more tasks the better.
  ➜ improves flexibility in the design.
- Limit the number of redundancy in data and computations.
  ➜ improves scalability for large problem.
- Tasks should be of similar sizes.
  ➜ improves load balancing.
- Number of tasks should depend on the size of the problem.
  ➜ improves efficiency.
- All decompositions should be considered.
  ➜ check for flexibility.

Communication

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 concurrents 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

- Load balancing of the communication operations.
  ➜ improves scalability.
- Small communication pattern.
  ➜ improves scalability.
- Communication are concurrents to computations.
  ➜ improves scalability.
- Computations are concurrents 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

- Communication costs are reduced.
- Replication of data preserved scalability.
- Replication of computation preserved performances.
- Tasks are load balanced in term of computation and communication.
- Scalability is preserved.

Affectation

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

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

- Solve the evolution of the temperature in a rod using a 1D approach. The evolution in its discrete form is given by

$$u_i^n = ru_{i-1}^{n-1} + (1-2r)u_i^{n-1} + ru_{i+1}^{n-1}$$

- Find the maximum in an array.

Exercise

Let us consider the matrix

$$K = \begin{bmatrix} -10 & -10 & 20 & -20 & -30 \\ 5 & -20 & 10 & 40 & 10 \\ 30 & -40 & 20 & -10 & -15 \\ -20 & 4 & -5 & 50 & 10 \\ 10 & -20 & 10 & -40 & 10 \end{bmatrix}$$

Problem: Find the maximum sub array of K, that is find the sub matrix in K such that the sum of the coefficients is maximum.

Question: design a parallel algorithm to solve this problem?