GP-GPU and High Performances Computing

Lecture 08 – Design of parallel program

December 1, 2023

Previsouly

- Organization of a computation with respect to data and architecture.
- Computations should carefully adapt to the architecture and maximize the usage of the ressources.
- Matrix multiply example.
- Reduce pattern.

Foster methodology

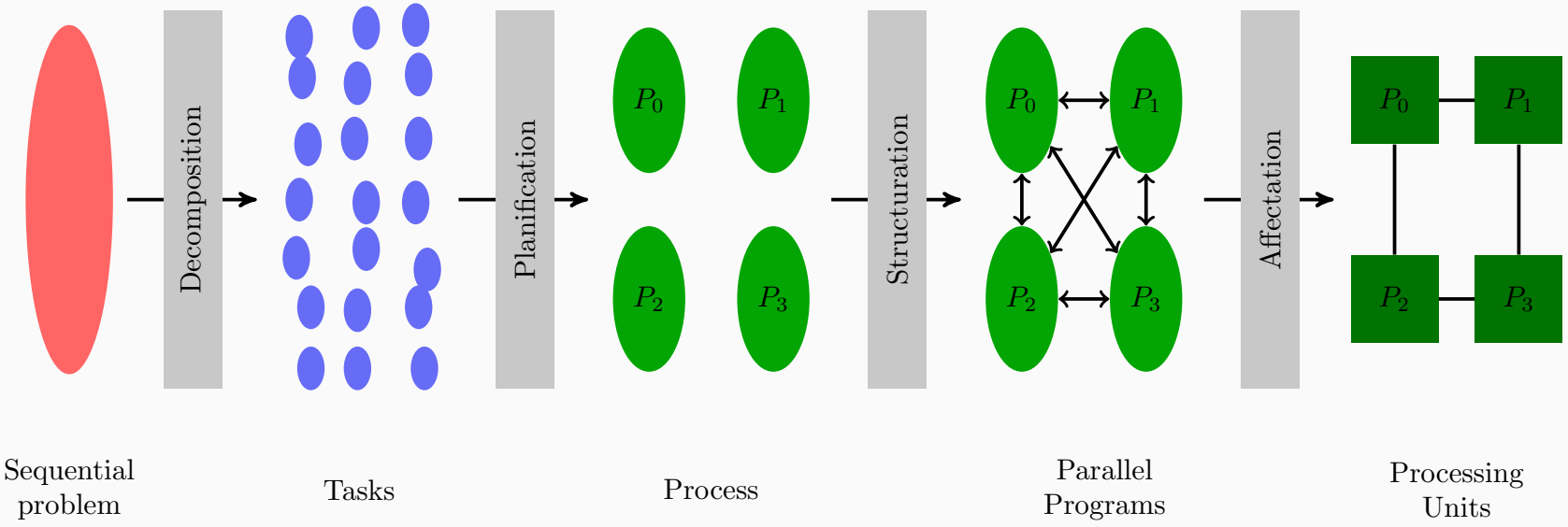
Choice criterion

- 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

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 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

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

Agglomeration

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.

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

1. Single Program Multiple Data 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.

Examples

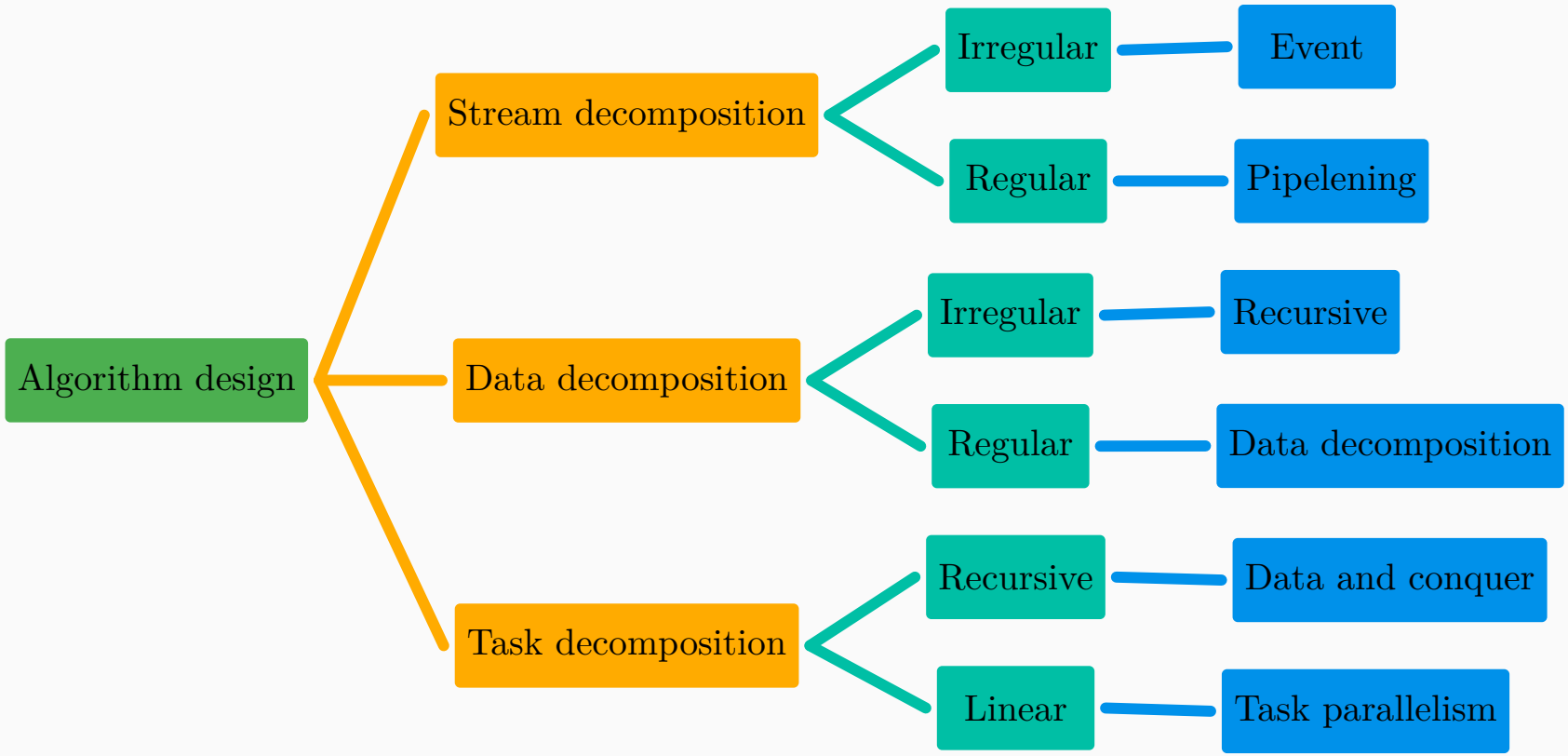
1. Solve the evolution of the temperature in a rod using a 1D approach. The evolution in its discrete form is given by

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

2. Find the maximum in an array.

Parallel patterns

Parallel Algorithms

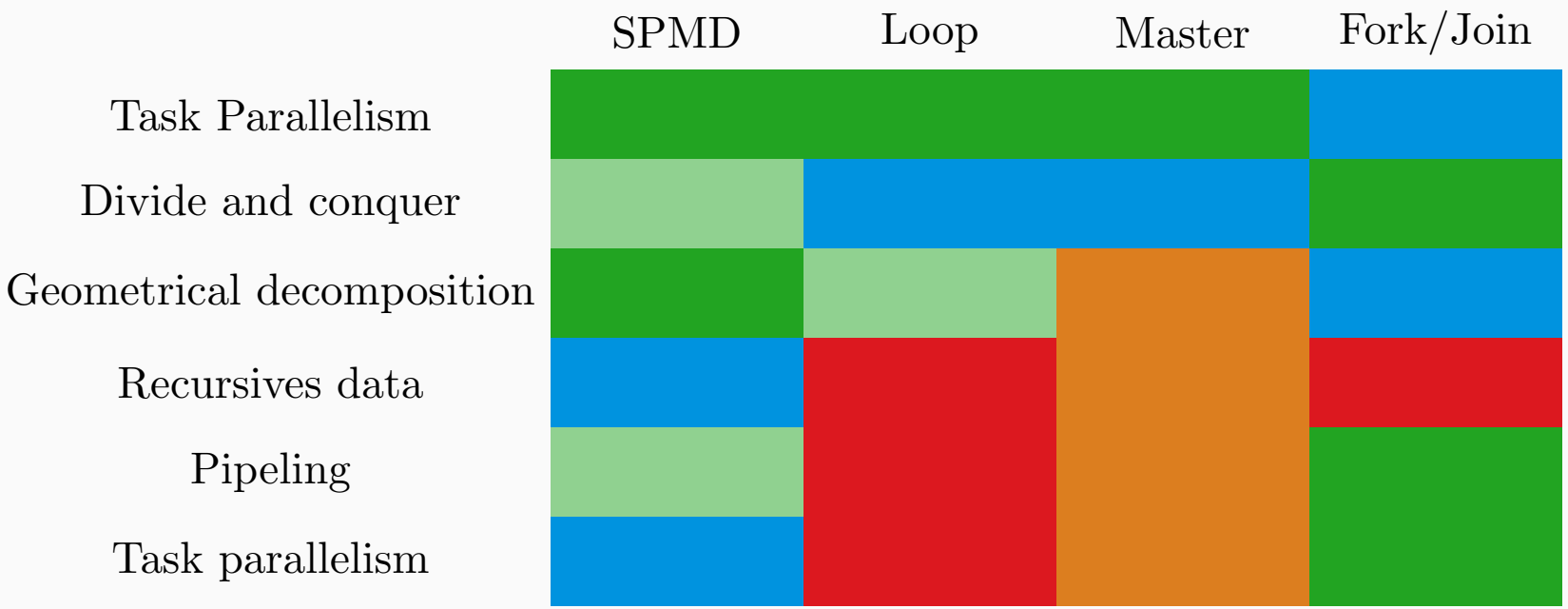


Parallel Patterns (1)

- SPMD — the same task runs on different processing units of a parallel platform. Each task run independently of the other and each stream of instruction may be different: branching, prediction, …
- Master/Slave — a main thread allocates tasks to slaves. In this pattern, the master may become a bottleneck or slaves may not have task to process (starvation).
- Manager/Worker — instead of the master allocating tasks, it keeps a pool of available tasks that the worker can request. In this pattern, the tasks are dynamically balanced between the workers.

Parallel Patterns (2)

- Thread pool — a pool of active threads is created by the operating system. A manager thread allocates tasks to the remaining threads removing them from the pool. The main advantage is that thread management is assigned to the OS, improving the performances. However, the operating system support increases as the resource usage increases.
- Join/Fork — a main thread fork itself into several independent threads, each executing its own tasks. It is well suited for data parallel problems with the problem of thread bottleneck for large system.
- Loop-Level — sequential loops are run in parallel, with the constraint that iterations should be independent.


SPMD
Loop
Master
Fork/Join
Task Parallelism
Divide and conquer
Geometrical decomposition
Recursives data
Pipeling
Task parallelism

Conclusion

Conclusion

- Design of parallel software
  - Decomposition
  - Communication
  - Agglomeration
  - Affectation