Designing parallel programs

Thoai Nam

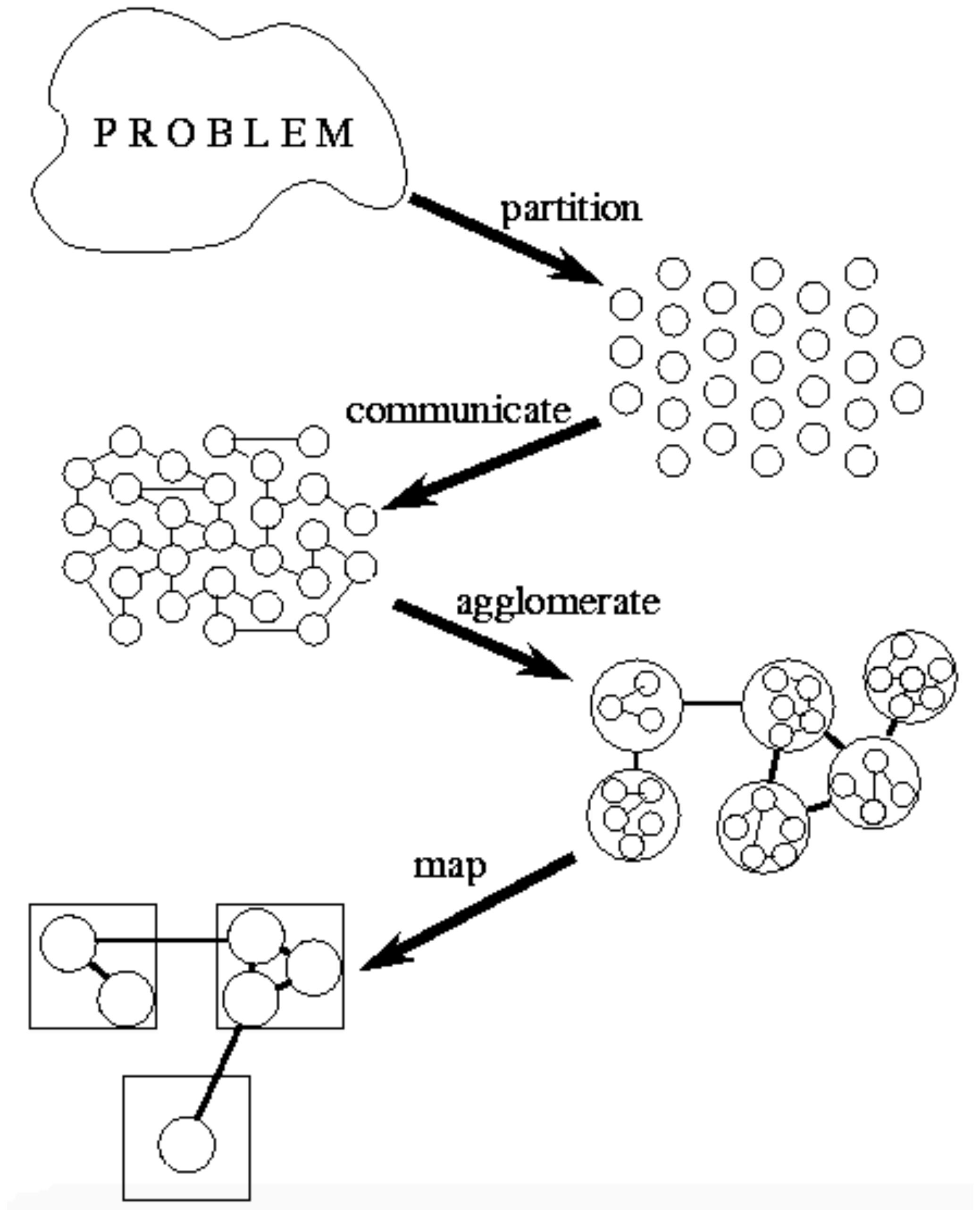
High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Technology

HCMC University of Technology

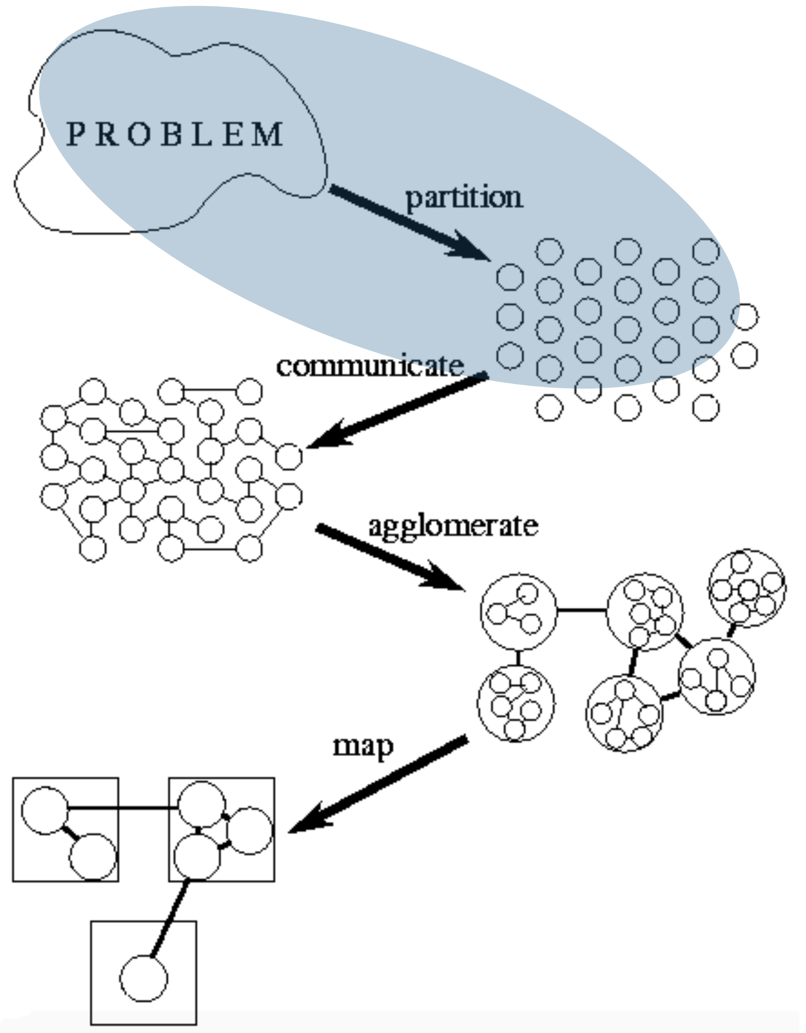
Introduction

- Ian Foster has proposed a 4-step process for designing parallel algorithms for machines that fit the task/channel model.
- It encourages the development of scalable algorithms by delaying machine-dependent considerations until the later steps.
- The 4 design steps are called:
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping



Partitioning

- Partitioning: dividing the computation and data into pieces
- Domain decomposition
  - Divide data into pieces
  - Determine how to associate computations with the data
  - Focuses on the largest and most frequently accessed data structure
- Functional decomposition
  - Divide computation into pieces
  - Determine how to associate data with the computations
  - This often yields tasks that can be pipelined.
  - SKIP, less important method

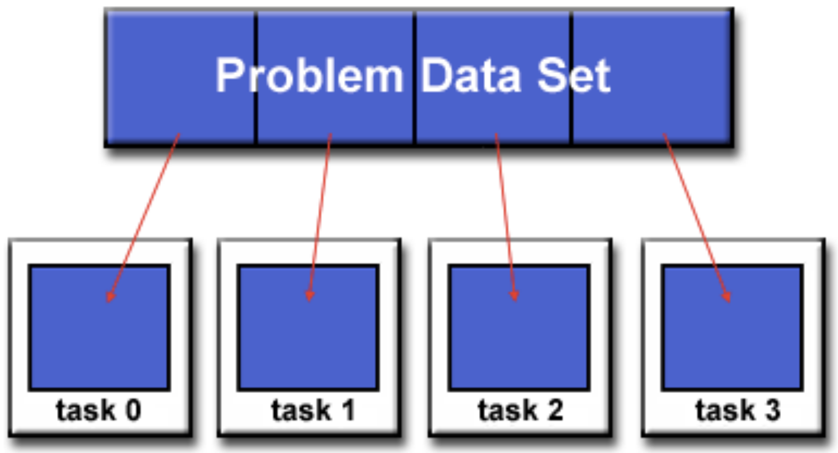


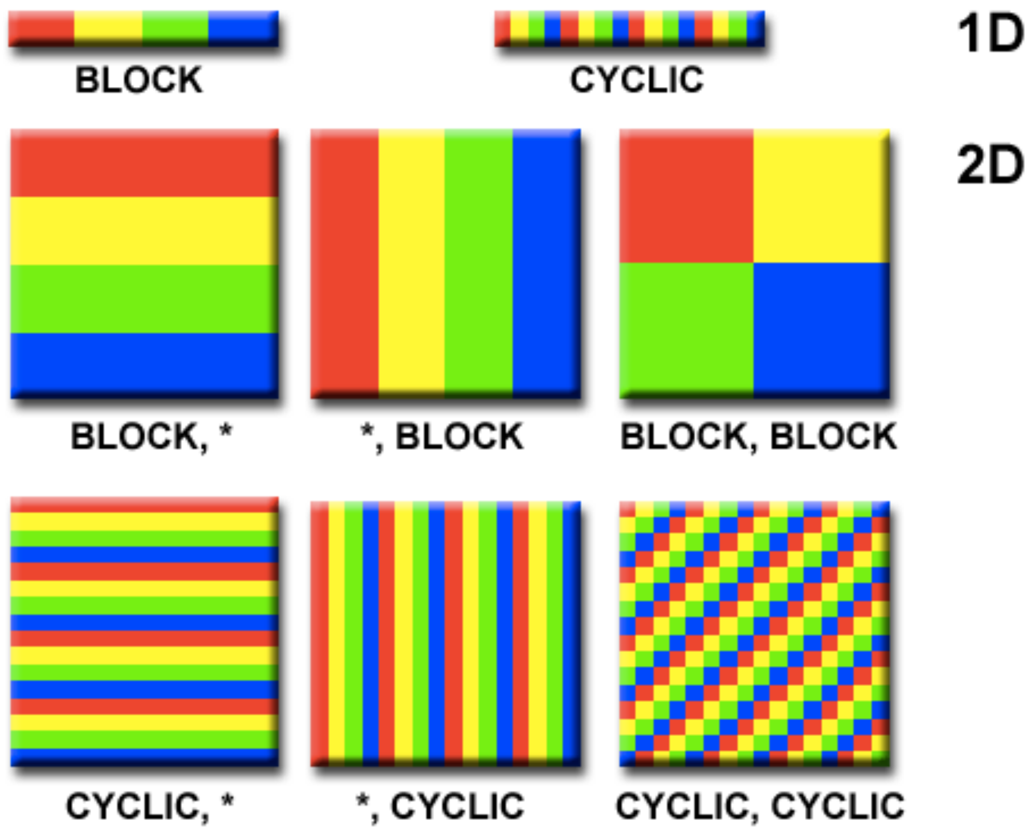
Goals for partition

- At least 10x more primitive tasks than processors in target computer
- Minimize redundant computations and redundant data storage
- Primitive tasks of roughly the same size
- Number of tasks an increasing function of problem size
- Remember – we are talking about MIMDs here which typically have a lot less processors than SIMDs.

Domain decompositions

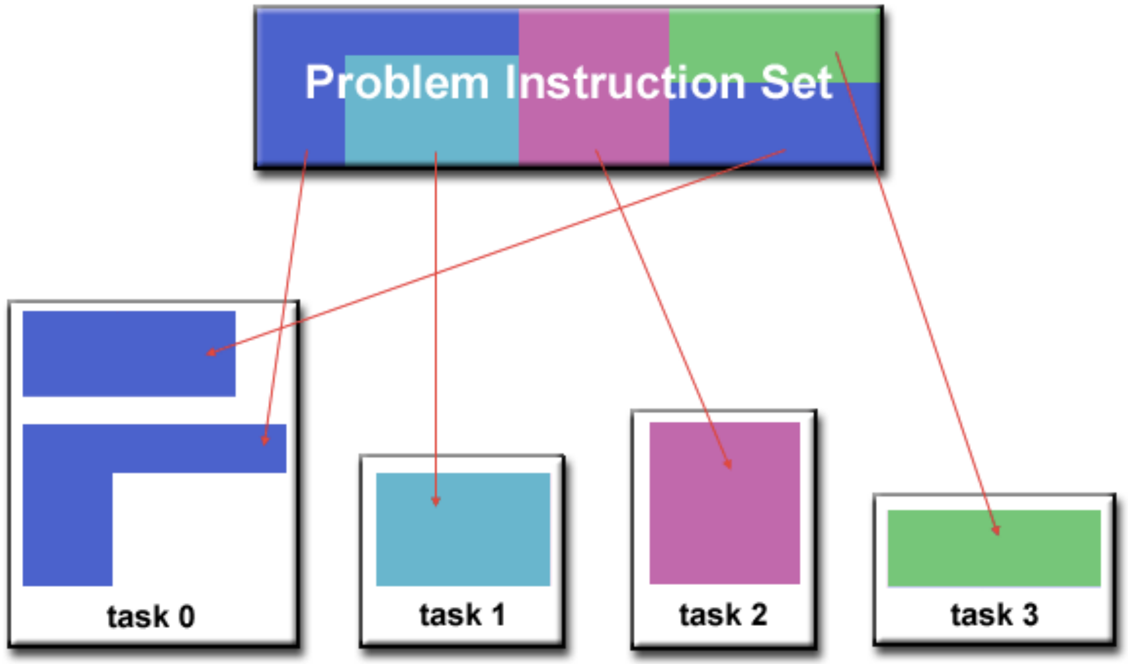
- In this type of partitioning, the data associated with a problem is decomposed
- Each parallel task then works on a portion of the data.





Functional decompositions

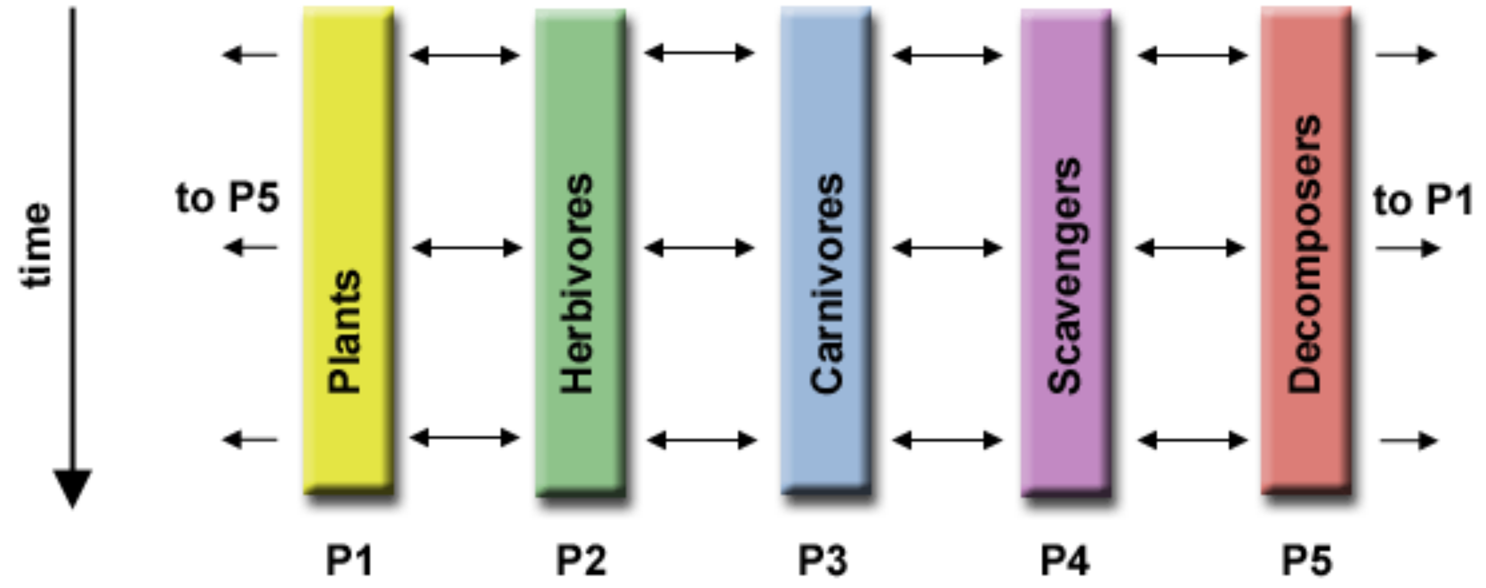
- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- The problem is decomposed according to the work that must be done
- Each task then performs a portion of the overall work



Functional decomposition lends itself well to problems that can be split into different tasks

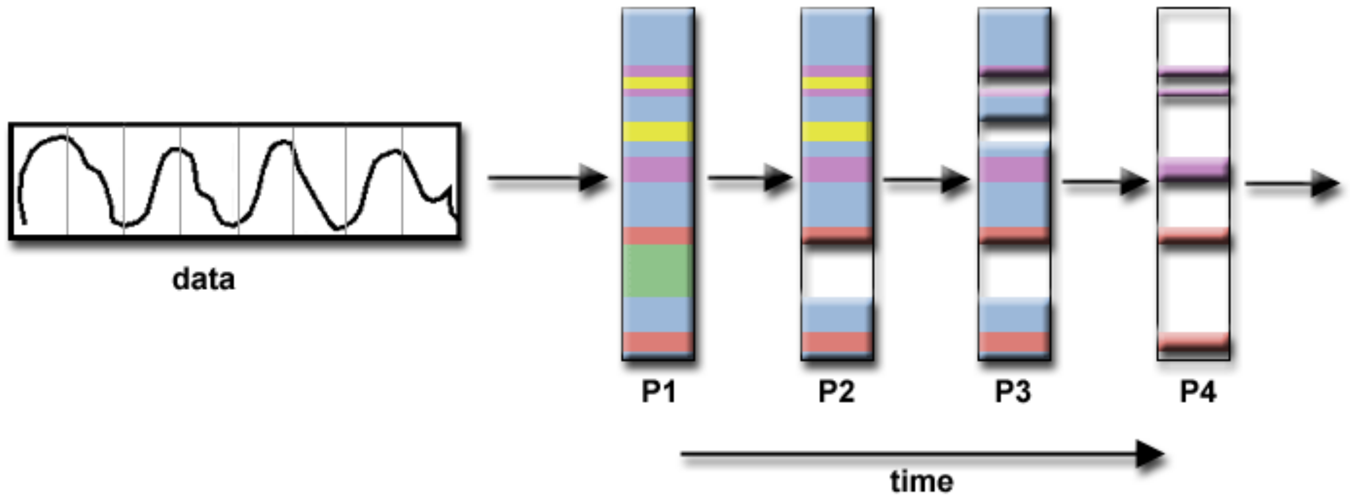
Ecosystem modeling

Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time step.



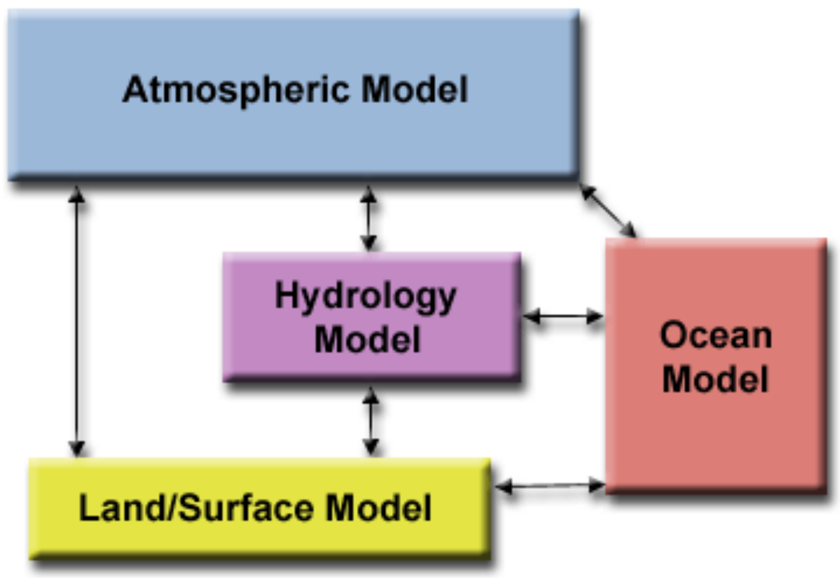
Signal processing

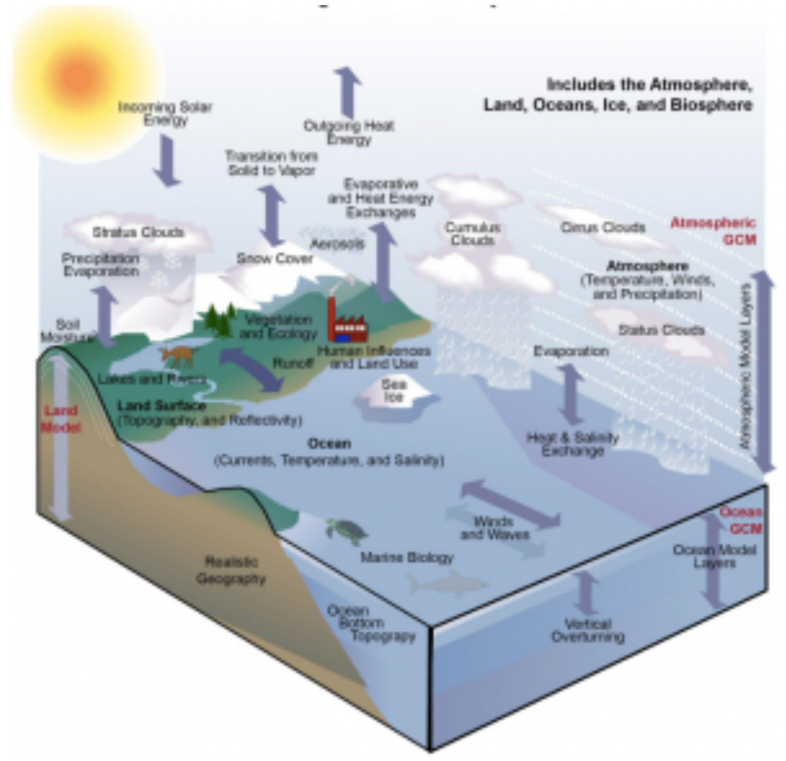
- An audio signal data set is passed through four distinct computational filters
- Each filter is a separate process
- The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy.



Climate modeling

- Each model component can be thought of as a separate task
- Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.

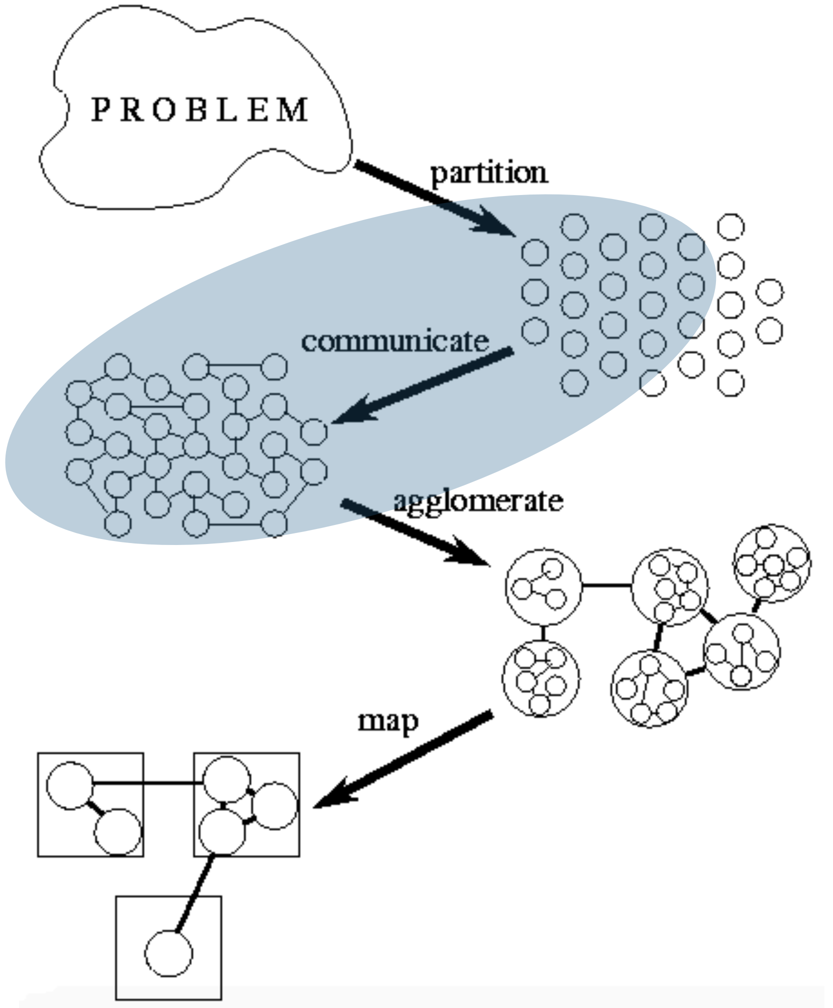




Combining domain & functional decompositions is common and natural

Communication

- Determine values passed among tasks
- There are two kinds of communication:
- Local communication
  - A task needs values from a small number of other tasks
  - Create channels illustrating data flow
- Global communication
  - A significant number of tasks contribute data to perform a computation
  - Don’t create channels for them early in design



Communication (2)

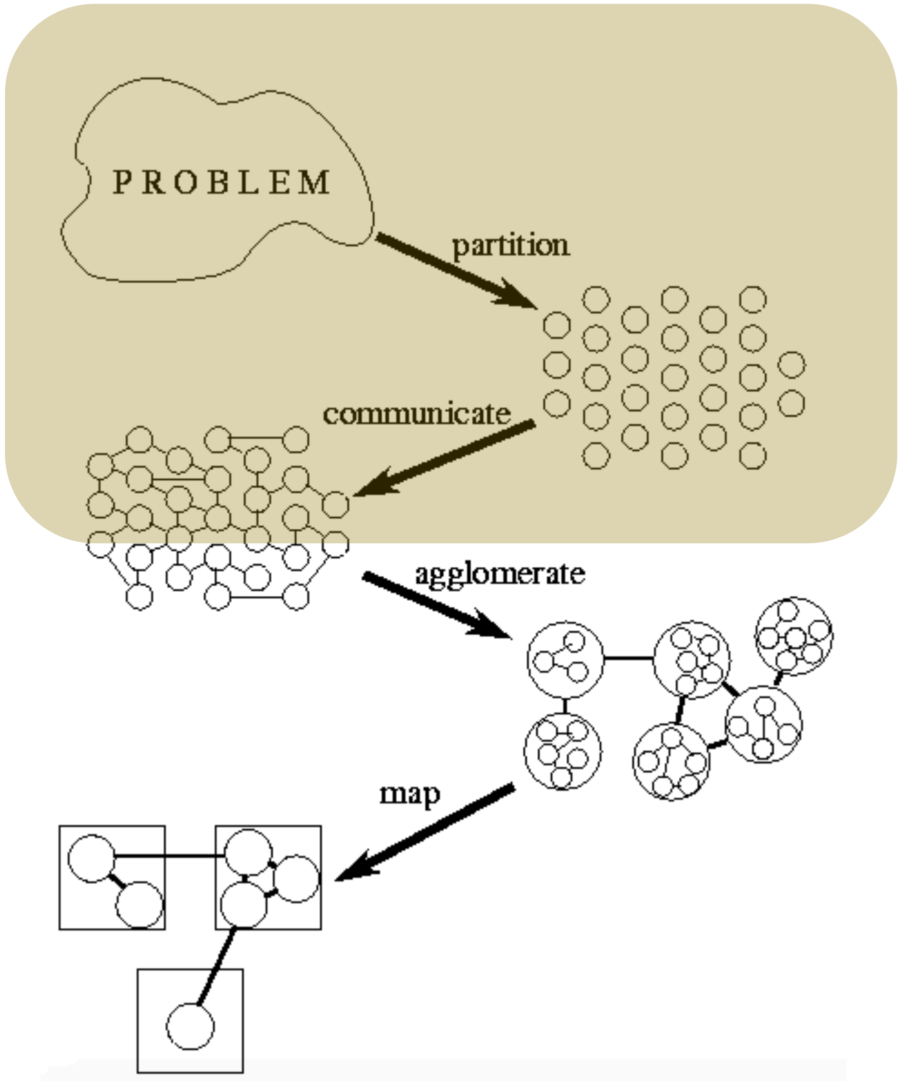
- Communications is part of the parallel computation overhead since it is something sequential algorithms do not have do
    - Costs larger if some (MIMD) processors have to be synchronized
- SIMD algorithms have much smaller communication overhead because
    - Much of the SIMD data movement is between the control unit and the PEs on broadcast/reduction circuits
        - ✧ especially true for associative
    - Parallel data movement along the interconnection network involves lockstep (i.e. synchronously) moves.

Communication checklist goals

- Communication operations should be balanced among tasks
- Each task communicates with only a small group of neighbors
- Tasks can perform communications concurrently
- Task can perform computations concurrently.

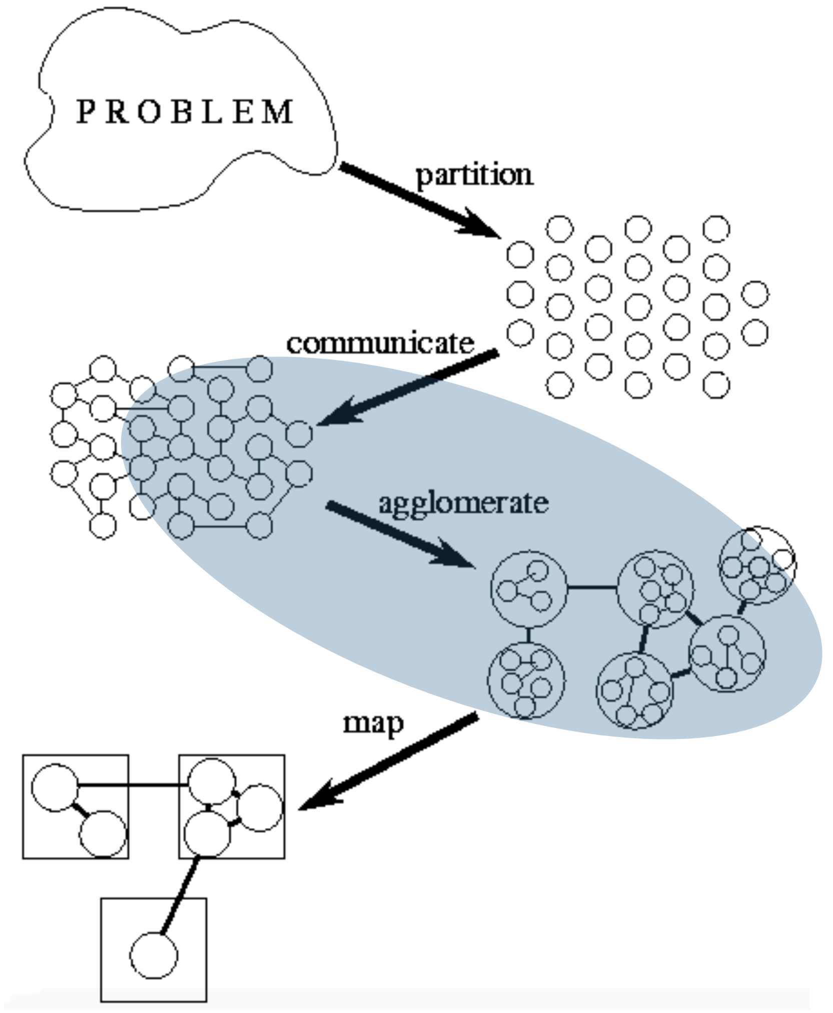
What we have hopefully at this point – and what we don’t have

- The first two steps look for parallelism in the problem
- However, the design obtained at this point probably does not map well onto a real machine
- If the number of tasks greatly exceed the number of processors, the overhead will be strongly affected by how the tasks are assigned to the processors
- Now we have to decide what type of computer we are targeting
  - Is it a centralized multiprocessor or a multicomputer?
  - What communication paths are supported
  - How must we combine tasks in order to map them effectively onto processors?



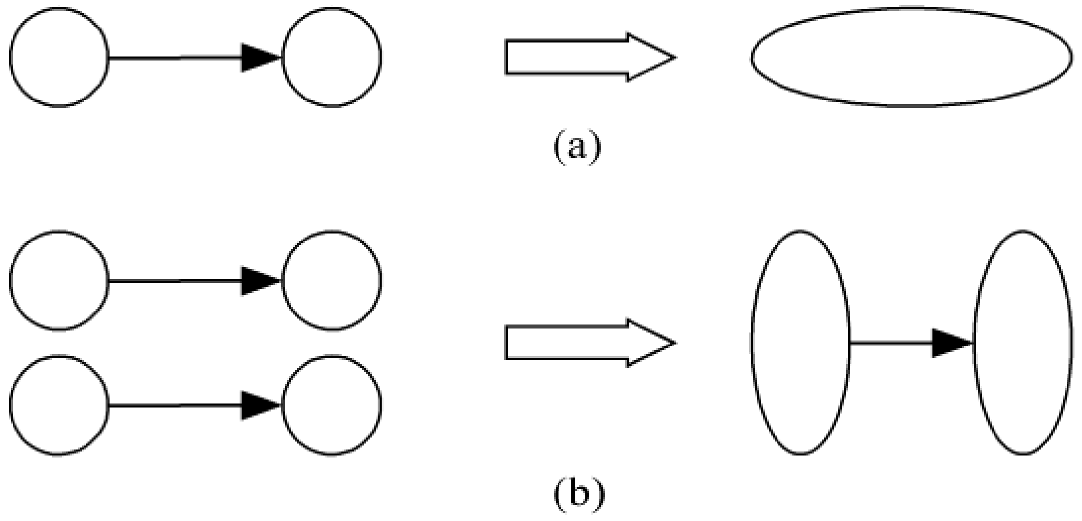
Agglomeration

- Agglomeration: Grouping tasks into larger tasks
- Goals
  - Improve performance
  - Maintain scalability of program
  - Simplify programming – i.e. reduce software engineering costs
  - to lower communication overhead
  - often to create one agglomerated task per processor
- By agglomerating primitive tasks that communicate with each other, communication is eliminated as the needed data is local to a processor.



Agglomeration can improve performance

- It can eliminate communication between primitive tasks agglomerated into consolidated task
- It can combine groups of sending and receiving tasks

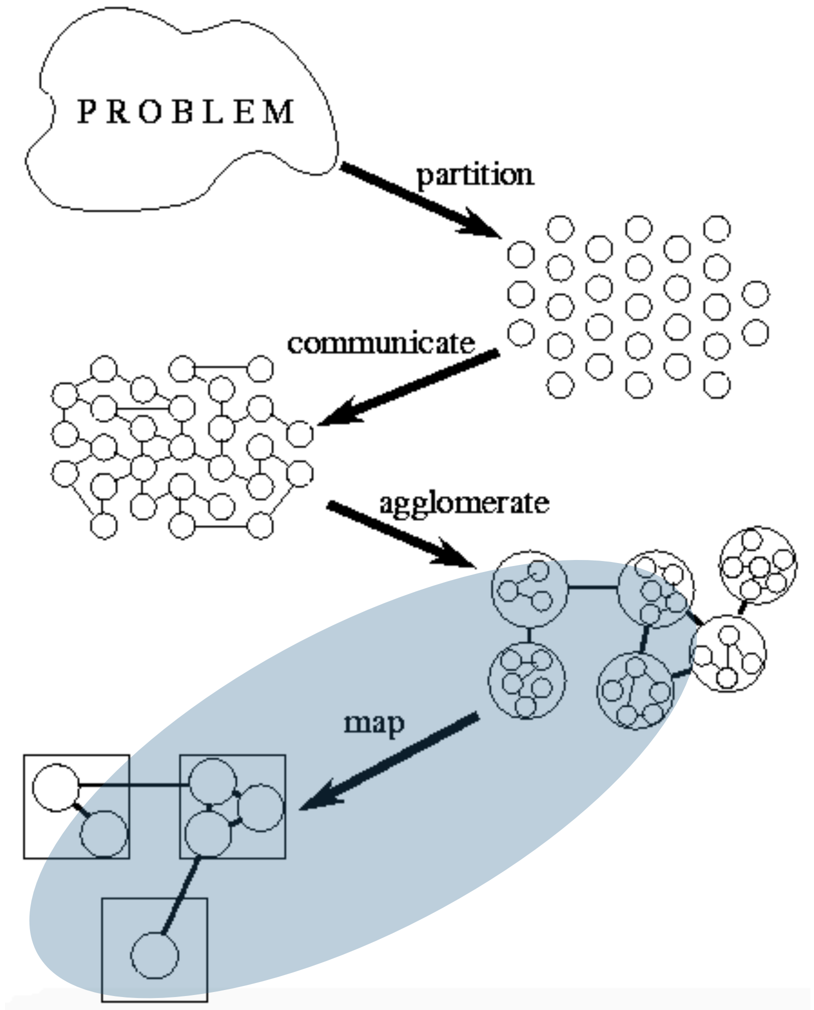


Agglomeration goals

- Replicated computations take less time than communications they replace
- Data replication does not affect scalability
- All agglomerated tasks have similar computational and communications costs
- Number of tasks increases with problem size
- Number of tasks suitable for likely target systems
- Tradeoff between agglomeration and code modifications costs is reasonable

Mapping

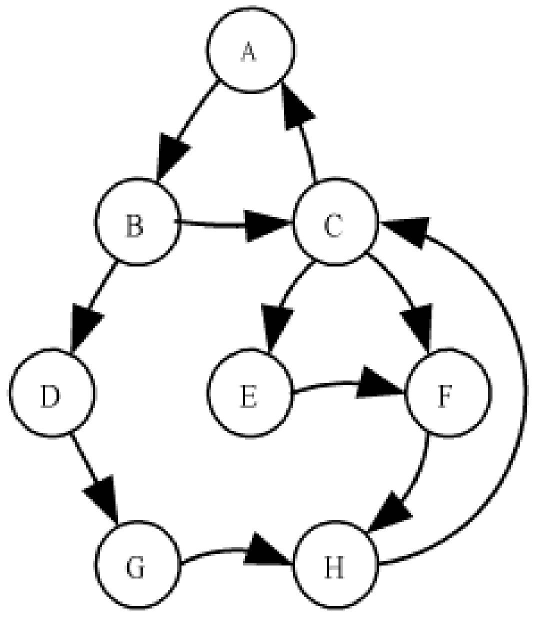
- Mapping: the process of assigning tasks to processors
- Centralized multiprocessor: mapping done by operating system
- Distributed memory system: mapping done by user
- Conflicting goals of mapping
  - Maximize processor utilization – i.e. the average percentage of time the system's processors are actively executing tasks necessary for solving the problem
  - Minimize interprocessor communication.



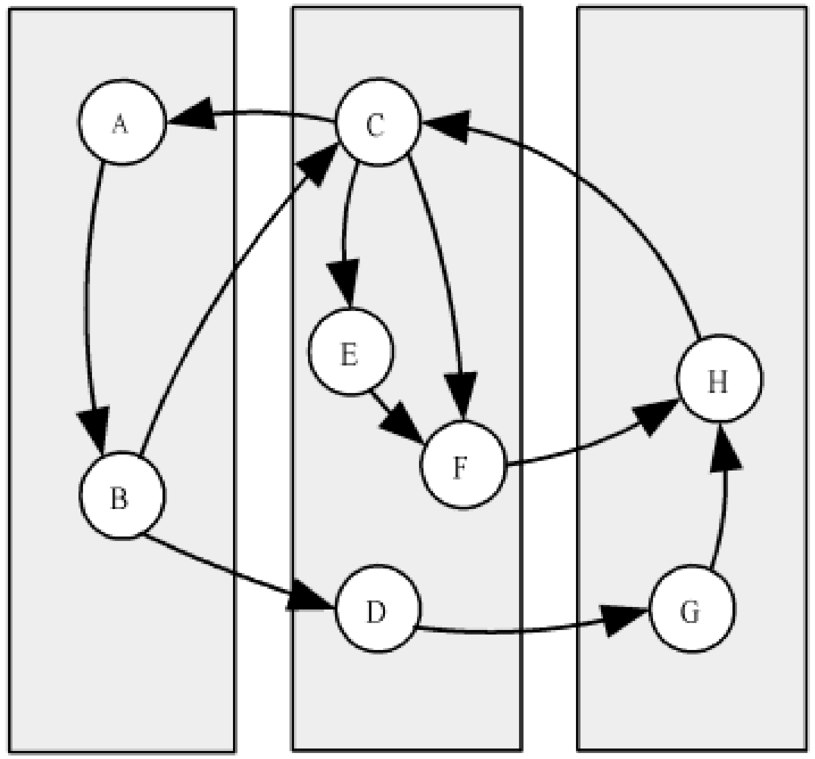
Mapping example

(a) is a task/channel graph showing the needed communications over channels.

(b) shows a possible mapping of the tasks to 3 processors.



(a)



(b)

If all tasks require the same amount of time and each CPU has the same capability, this mapping would mean the middle processor will take twice as long as the other two.

Optimal mapping

- Optimality is with respect to processor utilization and interprocessor communication
- Finding an optimal mapping is NP-hard
- Must rely on heuristics applied either manually or by the operating system
- It is the interaction of the processor utilization and communication that is important
- For example, with p processors and n tasks, putting all tasks on 1 processor makes interprocessor communication zero, but utilization is $1/p$.

A Mapping decision tree
(Quinn's suggestions)

Static number of tasks

- Structured communication
  - Constant computation time per task
    - Agglomerate tasks to minimize communications
    - Create one task per processor
  - Variable computation time per task
    - Cyclically map tasks to processors
- Unstructured communication
  - Use a static load balancing algorithm

Dynamic number of tasks

- Frequent communication between tasks
  - Use a dynamic load balancing algorithm
- Many short-lived tasks. No internal communication
  - Use a run-time task-scheduling algorithm

Mapping checklist goals

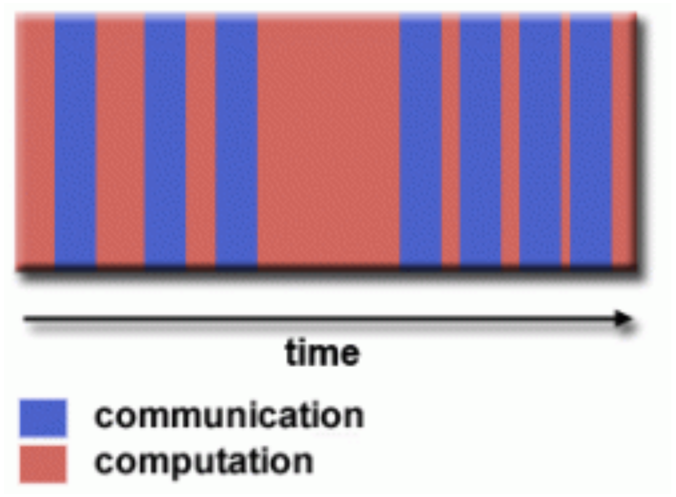
- Consider designs based on one task per processor and multiple tasks per processor
- Evaluate static and dynamic task allocation
- If dynamic task allocation chosen, the task allocator (i.e., manager) is not a bottleneck to performance
- If static task allocation chosen, ratio of tasks to processors is at least 10:1

Granularity

- Computation / Communication ratio
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication
  - Periods of computation are typically separated from periods of communication by synchronization events.

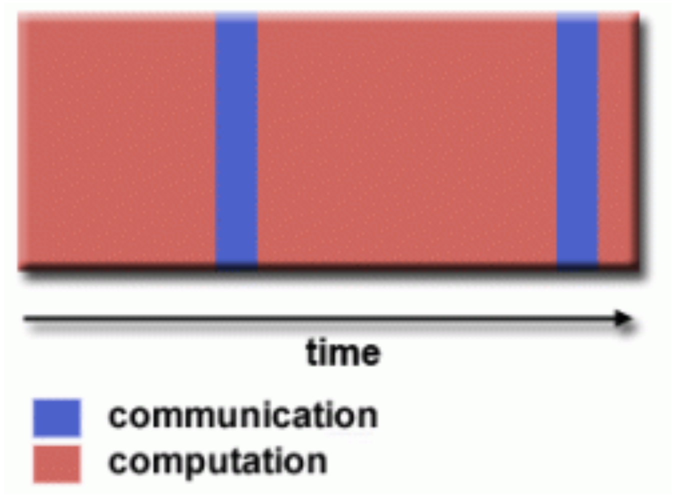
Fine-grain parallelism

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



Coarse-grain parallelism

- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently.



Which is best?

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity
- Fine-grain parallelism can help reduce overheads due to load imbalance.