Parallel Computing

2012

Slides credit: M. Quinn book (chapter 3 slides), A Grama book (chapter 3 slides)

Parallel Algorithm Design

Outline

- Computational Model
- Design Methodology
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping
- Example

Computational Model

Task: sequential program and its local storage

Parallel computation: two or more tasks executing concurrently

Communication channel: link between two tasks over which messages can be sent and received

send is nonblocking : sending task resumes execution immediately

receive is blocking : receiving task blocks execution until requested message is available

Example: Laplace Equation in 1-D

Consider Laplace equation in 1-D

$$u''(t) = 0$$

on interval $a < t < b$ with BC

$u(a) = \alpha$, $u(b) = \beta$

Seek approximate solution values $u_i \approx u(t_i)$ at mesh points

$t_i = a + ih$, $i = 0, \ldots, n + 1$, where

$h = (b-a)/(n + 1)$

Example: Laplace Equation in 1-D

- Finite difference approximation

$$u''(t_i) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$$

- yields tridiagonal system of algebraic equations

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = 0, \quad i = 1, \ldots, n,$$

  for u_i, i = 1, …., n, where $u_0 = \alpha$ and $u_{n+1} = \beta$

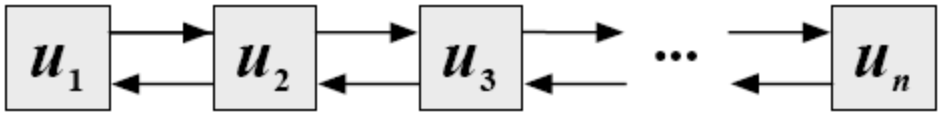
- Starting from initial guess $u^{(0)}$, compute Jacobi iterates

$$u_i^{(k+1)} = \frac{u_{i-1}^{(k)} + u_{i+1}^{(k)}}{2}, \quad i = 1, \ldots, n,$$

  for k = 1,…. until convergence

Example: Laplace Equation in 1-D

- Define n tasks, one for each u_i, i = 1 , . . . , n
- Task i stores initial value of u_i and updates it at each iteration until convergence
- To update u_i, necessary values of u_{i-1} and u_{i+1} obtained from neighboring tasks i-1 and i+1



- Tasks 1 and n determine u_0 and u_{n+1} from BC

Example: Laplace Equation in 1-D

initialize u_i
for $k = 1, \ldots$
    if $i > 1$, send u_i to task $i - 1$ { send to left neighbor }
    if $i < n$, send u_i to task $i + 1$ { send to right neighbor }
    if $i < n$, recv u_{i+1} from task $i + 1$ { receive from right neighbor }
    if $i > 1$, recv u_{i-1} from task $i - 1$ { receive from left neighbor }
    wait for sends to complete
    $u_i = (u_{i-1} + u_{i+1})/2$ { update my value }
end

Mapping Tasks to Processors

- Tasks must be assigned to physical processors for execution
- Tasks can be mapped to processors in various ways, including multiple tasks per processor
- Semantics of program should not depend on number of processors or particular mapping of tasks to processors
- Performance usually sensitive to assignment of tasks to processors due to concurrency, workload balance, communication patterns, etc
- Computational model maps naturally onto distributed-memory multicomputer using message passing

Other Models of Parallel Computation

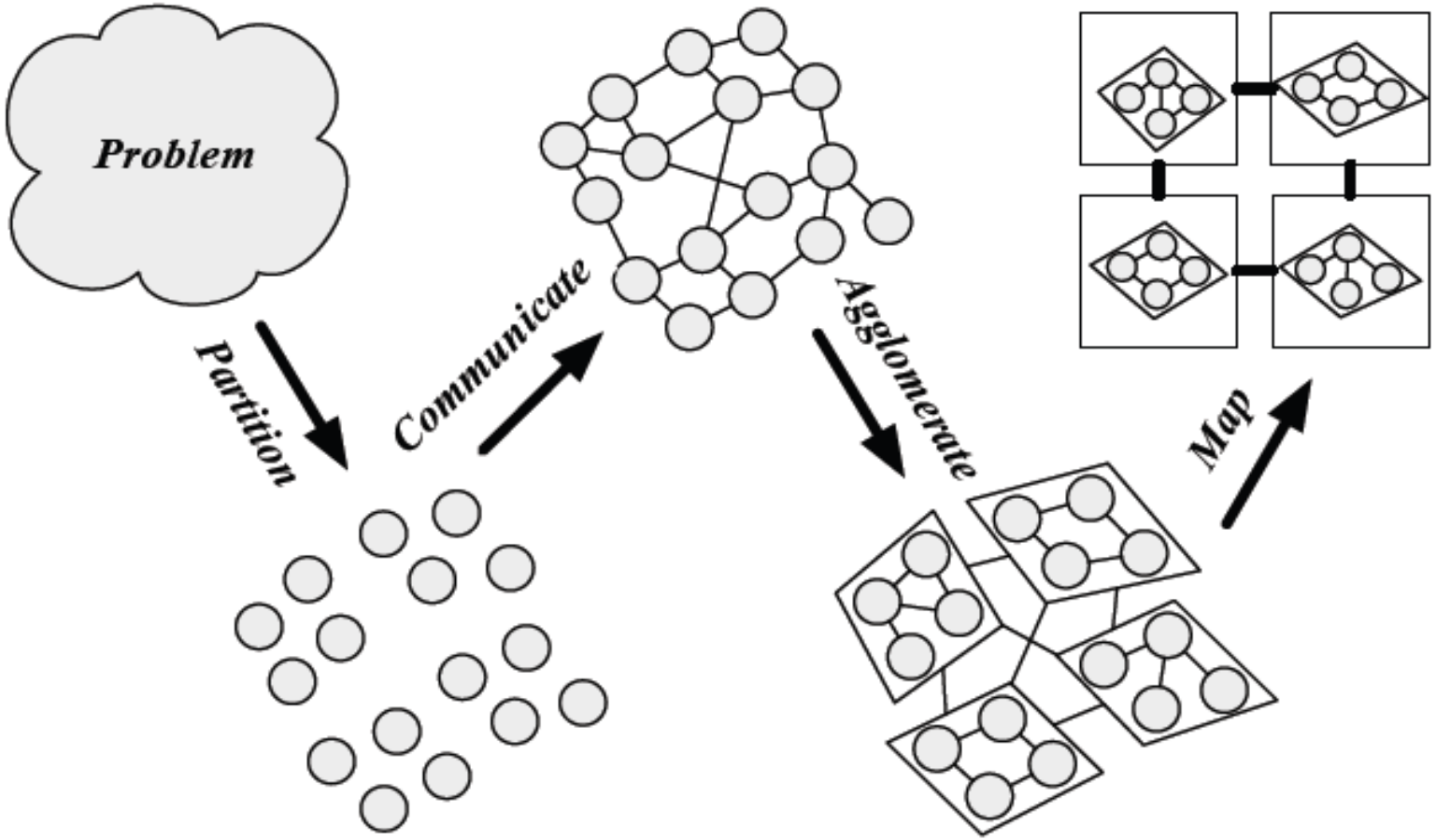
- PRAM — Parallel Random Access Machine
- LogP — Latency/Overhead/Gap/Processors
- BSP — Bulk Synchronous Parallel
- CSP — Communicating Sequential Processes
- Linda — Tuple Space
- and many others

Refer: http://www.ida.liu.se/~chrke/papers/modelsurvey.pdf

Four-Step Design Methodology

- Partition : Decompose problem into fine-grain tasks, maximizing number of tasks that can execute concurrently
- Communicate : Determine communication pattern among fine-grain tasks, yielding task graph with fine-grain tasks as nodes and communication channels as edges
- Agglomerate : Combine groups of fine-grain tasks to form fewer but larger coarse-grain tasks, thereby reducing communication requirements
- Map : Assign coarse-grain tasks to processors, subject to tradeoffs between communication costs and concurrency

Four-Step Design Methodology



Graph Embeddings

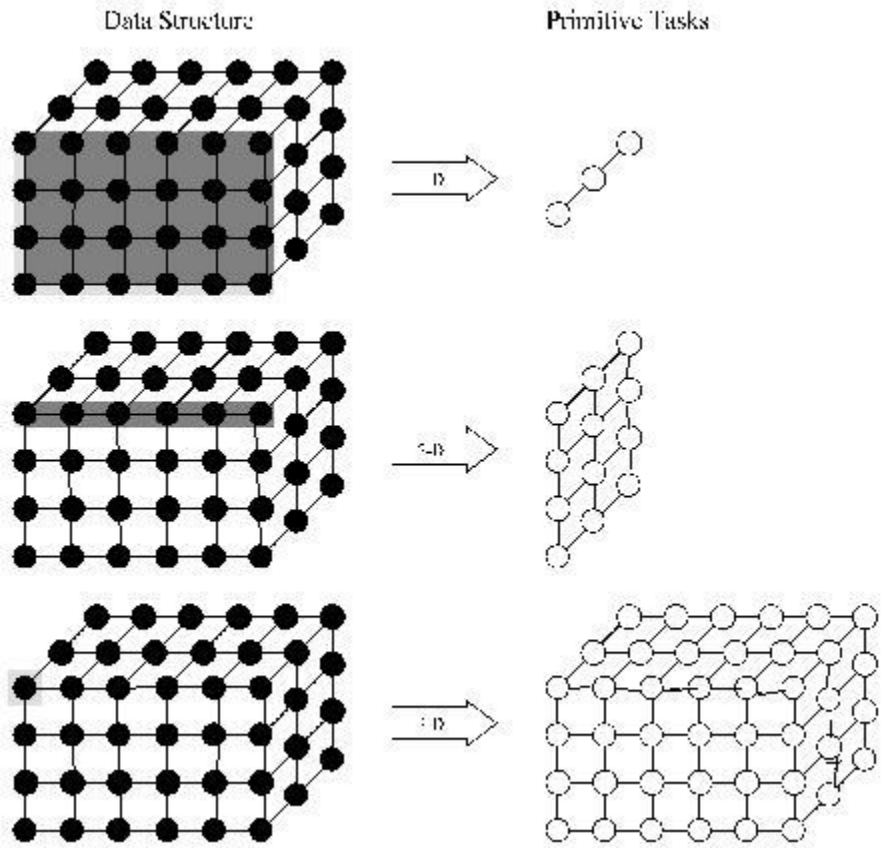
- Target network may be virtual network topology, with nodes usually called processes rather than processors
- Overall design methodology is composed of sequence of graph embeddings:
  - fine-grain task graph to coarse-grain task graph
  - coarse-grain task graph to virtual network graph
  - virtual network graph to physical network graph
- Depending on circumstances, one or more of these embeddings may be skipped
- Target system may automatically map processes of virtual network topology to processors of physical network

Partitioning Strategies

- 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
- Independent tasks : subdivide computation into tasks that do not depend on each other (*embarrassingly parallel*)
- Array parallelism : simultaneous operations on entries of vectors, matrices, or other arrays
- Divide-and-conquer : recursively divide problem into tree-like hierarchy of subproblems
- Pipelining : break problem into sequence of stages for each of sequence of objects

Please post examples for each on piazza.

Example Domain Decompositions



Think of the primitive tasks as processors.

In 1st, each 2D slice is mapped onto one processor of a system using 3 processors.

In second, a 1D slice is mapped onto a processor.

In last, an element is mapped onto a processor

The last leaves more primitive tasks and is usually preferred.

Desirable Properties of Partitioning

- Maximum possible concurrency in executing resulting tasks
- Many more tasks than processors
- Number of tasks, rather than size of each task, grows as overall problem size increases
- Tasks reasonably uniform in size
- Redundant computation or storage avoided

Communication Patterns

- Communication pattern determined by data dependences among tasks: because storage is local to each task, any data stored or produced by one task and needed by another must be communicated between them
- Communication pattern may be
  - local or global
  - structured or random
  - persistent or dynamically changing
  - synchronous or sporadic

Desirable Properties of Communication

- Frequency and volume minimized
- Highly localized (between neighboring tasks)
- Reasonably uniform across channels
- Network resources used concurrently
- Does not inhibit concurrency of tasks
- Overlapped with computation as much as possible

What We Have Hopefully at This Point – and What We Don't Have

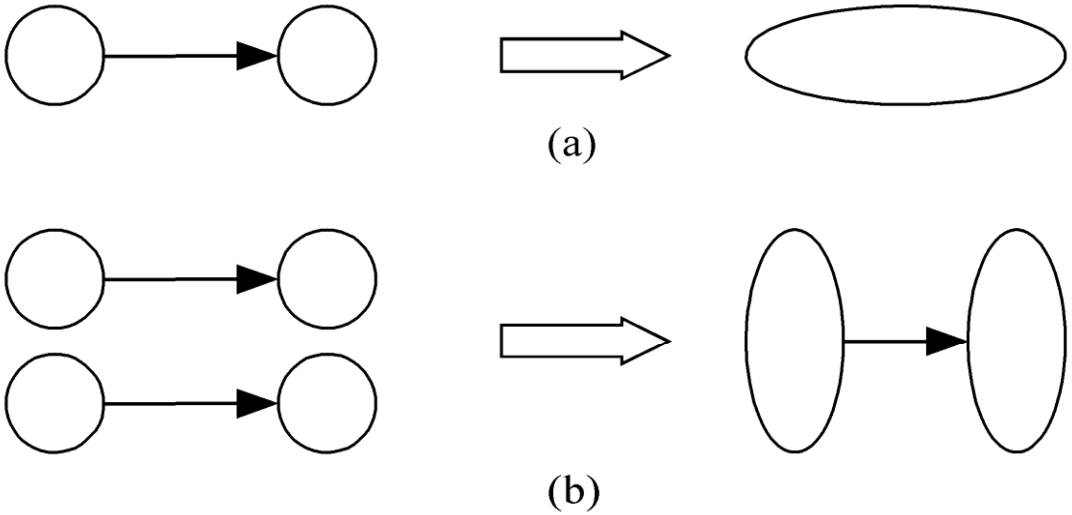
- The first two steps look for parallelism in the problem.
- However, the design obtained at this point probably doesn't map well onto a real machine.
- If the number of tasks greatly exceed the number of processors, the overhead will be strongly affected by how the tasks are assigned to the processors.
- 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.
- In MPI programming, a goal is
  - 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



Scalability

- Assume we are manipulating a 3D matrix of size 8 x 128 x 256 and
  - Our target machine is a centralized multiprocessor with 4 CPUs.
- Suppose we agglomerate the 2nd and 3rd dimensions. Can we run on our target machine?
  - Yes- because we can have tasks which are each responsible for a 2 x 128 x 256 submatrix.
  - Suppose we change to a target machine that is a centralized multiprocessor with 8 CPUs. Could our previous design basically work.
  - Yes, because each task could handle a 1 x 128 x 256 matrix.

Scalability

  - However, what if we go to more than 8 CPUs? Would our design change if we had agglomerated the 2nd and 3rd dimension for the 8 x 128 x 256 matrix?
  - Yes.
- This says the decision to agglomerate the 2nd and 3rd dimension in the long run has the drawback that the code portability to more CPUs is impaired.

Reducing Software Engineering Costs

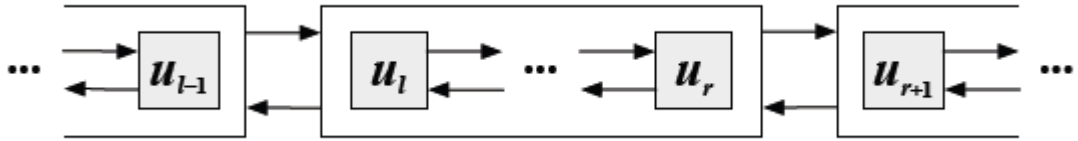
- Software Engineering – the study of techniques to bring very large projects in on time and on budget.
- One purpose of agglomeration is to look for places where existing sequential code for a task might exist,
- Use of that code helps bring down the cost of developing a parallel algorithm from scratch.

Agglomeration Checklist for Checking the Quality of the Agglomeration

- Locality of parallel algorithm has increased
- Replicated computations take less time than communications they replace
- Data replication doesn't affect scalability
- All the 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

Example: Laplace Equation in 1-D

- Combine groups of consecutive mesh points t_i and corresponding solution values u_i into coarse-grain tasks, yielding p tasks, each with n/p of u_i values



- Communication is greatly reduced, but u_i values within each coarse-grain task must be updated sequentially

Example: Laplace Equation in 1-D

initialize $u_l, \ldots, u_r$
for $k = 1, \ldots$
    if $j > 1$, send u_l to task $j - 1$ { send to left neighbor }
    if $j < p$, send u_r to task $j + 1$ { send to right neighbor }
    if $j < p$, recv u_{r+1} from task $j + 1$ { receive from right neighbor }
    if $j > 1$, recv u_{l-1} from task $j - 1$ { receive from left neighbor }
    for $i = l$ **to** r
        $\bar{u}_i = (u_{i-1} + u_{i+1})/2$ { update local values }
    end
    wait for sends to complete
    $\boldsymbol{u} = \bar{\boldsymbol{u}}$
end

Overlapping Communication and Computation

- Updating of solution values u_i is done only after all communication has been completed, but only two of those values actually depend on awaited data
- Since communication is often much slower than computation, initiate communication by sending all messages first, then update all “interior” values while awaiting values from neighboring tasks
- Much (possibly all) of updating can be done while task would otherwise be idle awaiting messages
- Performance can often be enhanced by overlapping communication and computation in this manner

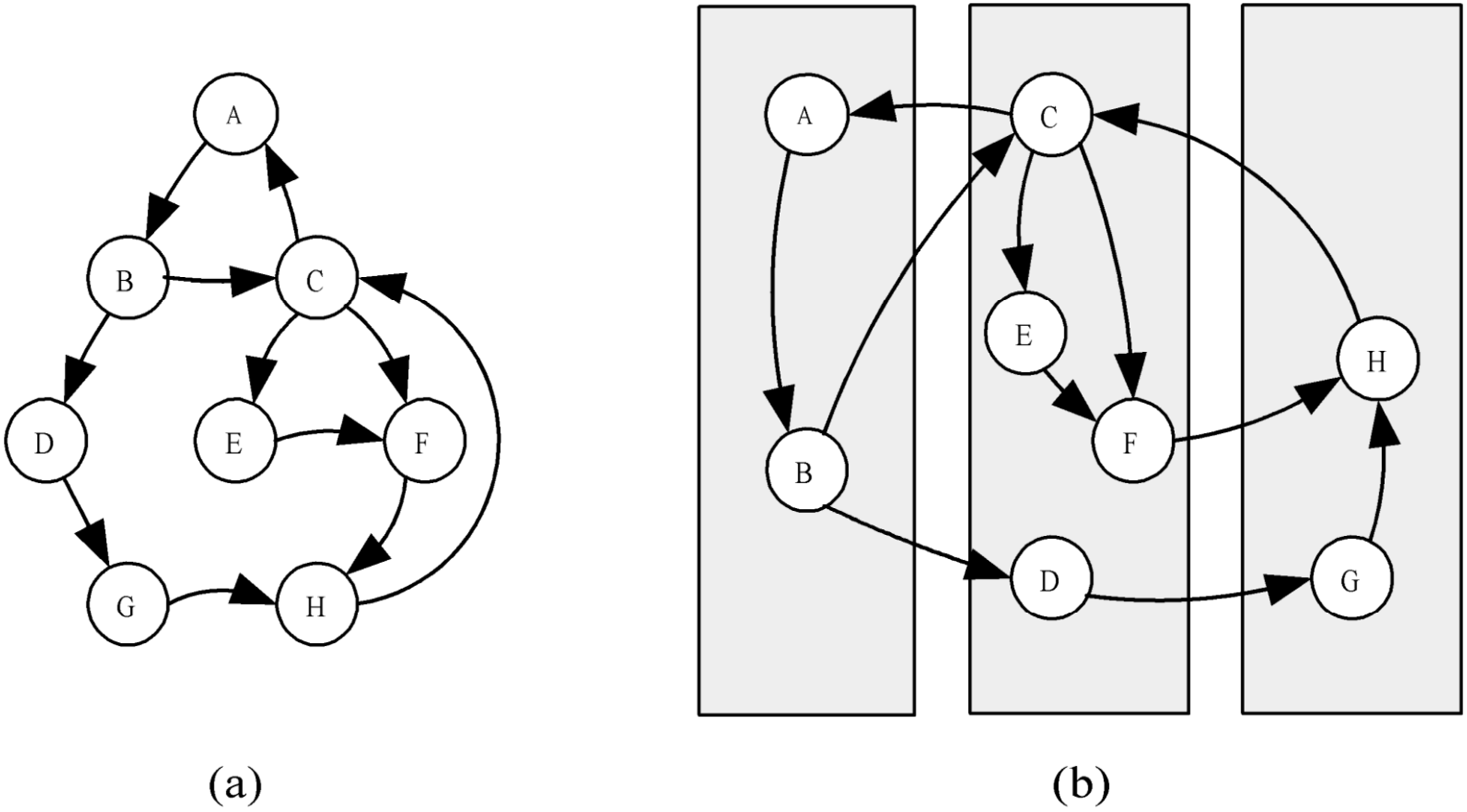
Example: Laplace Equation in 1-D

initialize $u_l, \ldots, u_r$
for $k = 1, \ldots$
  if $j > 1$, send u_l to task $j - 1$ { send to left neighbor }
  if $j < p$, send u_r to task $j + 1$ { send to right neighbor }
  for $i = l + 1$ **to** $r - 1$
    $\bar{u}_i = (u_{i-1} + u_{i+1})/2$ { update local values }
  end
  if $j < p$, recv u_{r+1} from task $j + 1$ { receive from right neighbor }
  $\bar{u}_r = (u_{r-1} + u_{r+1})/2$ { update local value }
  if $j > 1$, recv u_{l-1} from task $j - 1$ { receive from left neighbor }
  $\bar{u}_l = (u_{l-1} + u_{l+1})/2$ { update local value }
  wait for sends to complete
  $\boldsymbol{u} = \bar{\boldsymbol{u}}$
end

Mapping

- As with agglomeration, mapping of coarse-grain tasks to processors should maximize concurrency, minimize communication, maintain good workload balance, etc
- But connectivity of coarse-grain task graph is inherited from that of fine-grain task graph, whereas connectivity of target interconnection network is independent of problem
- Communication channels between tasks may or may not correspond to physical connections in underlying interconnection network between processors

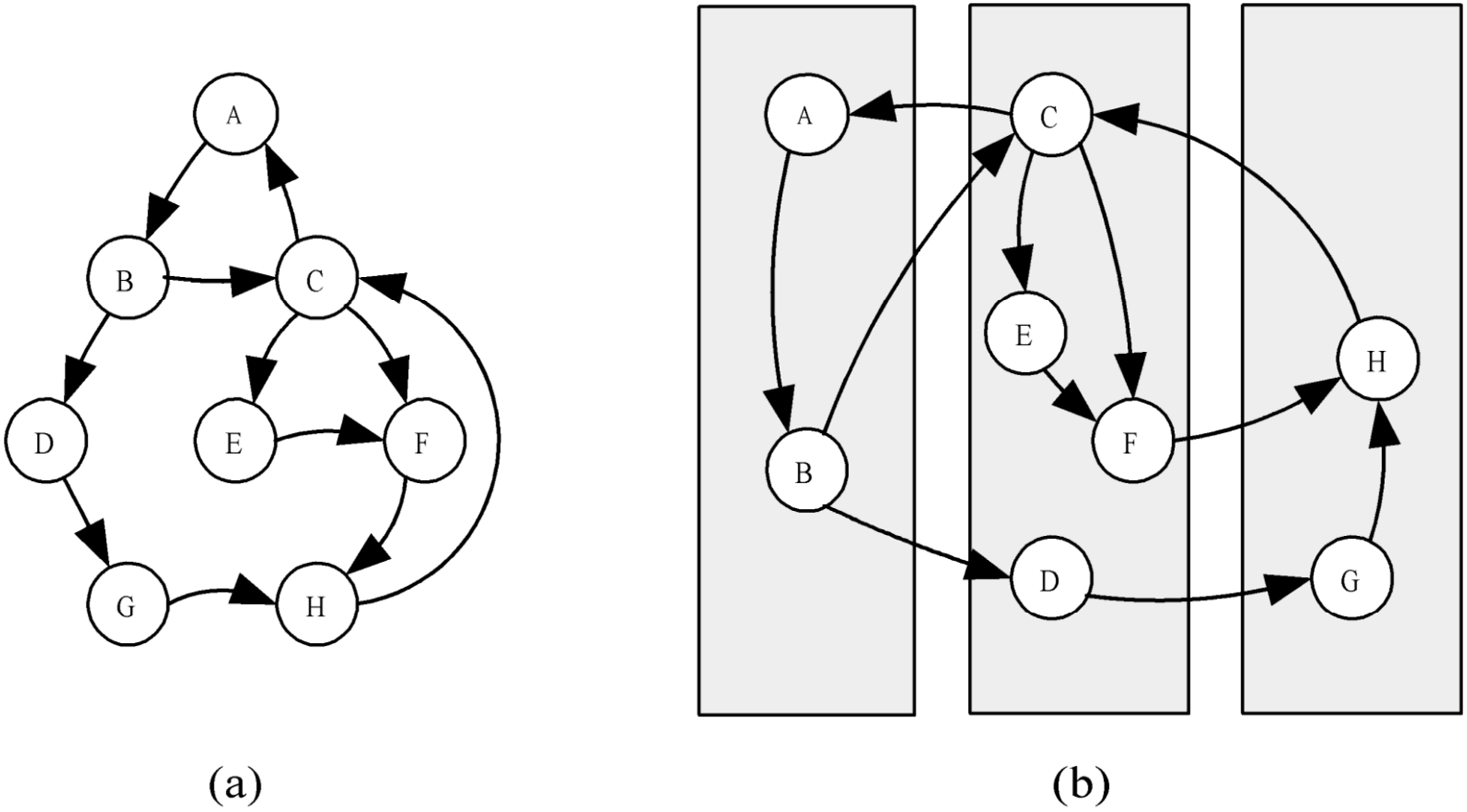
Mapping Example



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

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

Mapping Example



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 – Details on pg 72)

- 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

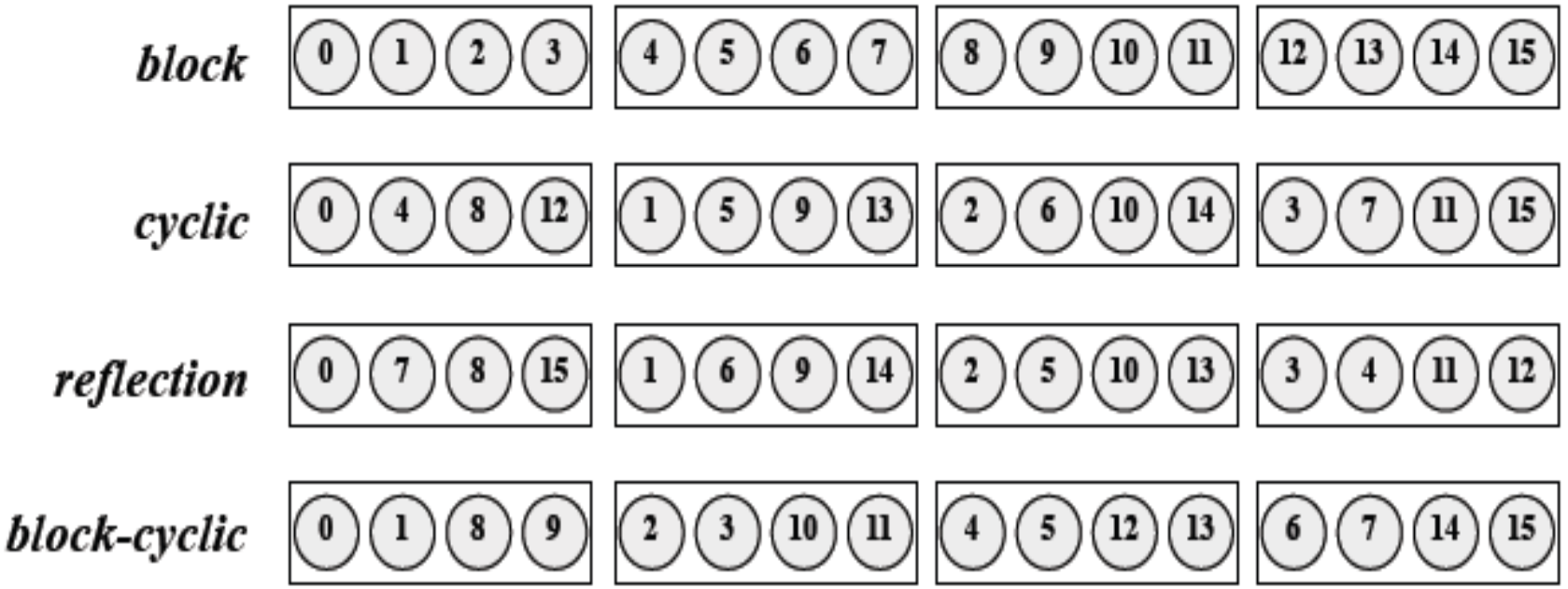
Mapping Checklist to Judge the Quality of a Mapping

- 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

Mapping Strategies

- With tasks and processors consecutively numbered in some ordering,
  - block mapping : blocks of n=p consecutive tasks are assigned to successive processors
  - cyclic mapping : task i is assigned to processor i mod p
  - reflection mapping : like cyclic mapping except tasks are assigned in reverse order on alternate passes
  - block-cyclic mapping and block-reflection mapping : blocks of tasks assigned to processors as in cyclic or reflection
- For higher-dimensional grid, these mappings can be applied in each dimension

Examples of Mappings



Dynamic Mapping

- If task sizes vary during computation or can't be predicted in advance, tasks may need to be reassigned to processors dynamically to maintain reasonable workload balance throughout computation
- To be beneficial, gain in load balance must more than offset cost of communication required to move tasks and their data between processors
- Dynamic load balancing usually based on local exchanges of workload information (and tasks, if necessary), so work diffuses over time to be reasonably uniform across processors

Task Scheduling

With multiple tasks per processor, execution of those tasks must be scheduled over time

For shared-memory, any idle processor can simply select next ready task from common pool of tasks

For distributed-memory, analogous approach is manager/worker paradigm, with manager dispatching tasks to workers

Manager/worker scales poorly, as manager becomes bottleneck, so hierarchy of managers and workers becomes necessary, or more decentralized scheme

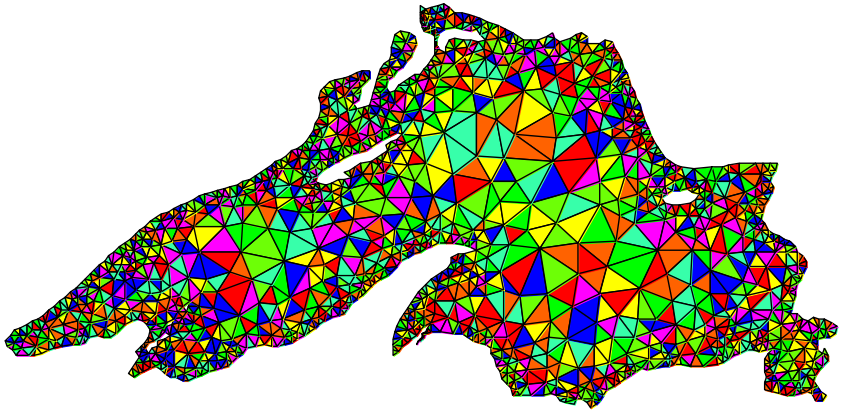
Task Scheduling

- For completely decentralized scheme, it can be difficult to determine when overall computation has been completed, so termination detection scheme is required
- With multithreading, task scheduling can conveniently be driven by availability of data: whenever executing task becomes idle awaiting data, another task is executed
- For problems with regular structure, it is often possible to determine mapping in advance that yields reasonable load balance and natural order of execution

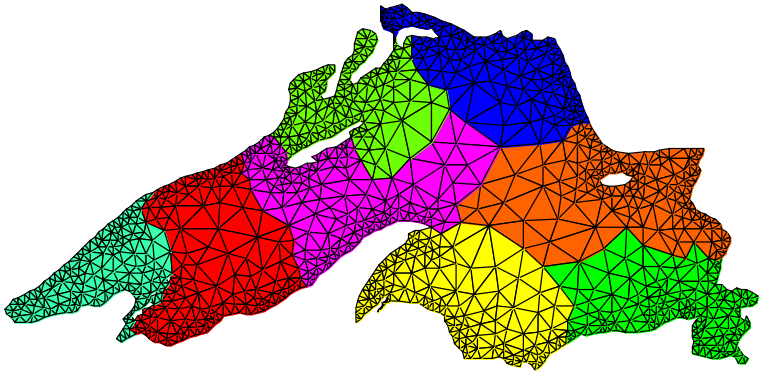
Graph Partitioning Dased Data Decomposition

- In case of sparse matrices, block decompositions are more complex.
- Consider the problem of multiplying a sparse matrix with a vector.
- The graph of the matrix is a useful indicator of the work (number of nodes) and communication (the degree of each node).
- In this case, we would like to partition the graph so as to assign equal number of nodes to each process, while minimizing edge count of the graph partition.

Partitioning the Graph of Lake Superior



Random Partitioning



Partitioning for minimum edge-cut.