Outline

1. Computational Model

2. Design Methodology
   - Partitioning
   - Communication
   - Agglomeration
   - Mapping

3. Example
**Computational Model**

- **Task**: a subset of the overall program, with a set of inputs and outputs

- **Parallel computation**: a program that executes two or more tasks concurrently

- **Communication channel**: connection between two tasks over which information is passed (messages are sent and received) periodically

For now we work with the following messaging semantics

- **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
  \[ u(a) = \alpha, \quad u(b) = \beta \]
- Seek approximate solution vector \( u \) such that
  \[ u_i \approx u(t_i) \quad \text{at mesh points} \quad t_i = a + ih, \forall i \in 0, \ldots, n + 1, \]
  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 \( u_i, \ i = 1, \ldots, n \), where \( u_0 = \alpha \) and \( u_{n+1} = \beta \)

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

\[ u^{(k+1)}_i = \frac{u^{(k)}_{i-1} + u^{(k)}_{i+1}}{2}, \quad i = 1, \ldots, n, \]

for \( k = 1, \ldots \) until convergence
Example: Laplace Equation in 1-D

- Define $n$ tasks, one for each $u_i$, $i = 1, \ldots, n$
- 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 $i - 1$
if $i < n$, send $u_i$ to task $i + 1$
if $i < n$, recv $u_{i+1}$ from task $i + 1$
if $i > 1$, recv $u_{i-1}$ from task $i - 1$
wait for sends to complete

$u_i = (u_{i-1} + u_{i+1})/2$

end

{ send to left neighbor }
{ send to right neighbor }
{ receive from right neighbor }
{ receive from left neighbor }
{ update my value }
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.
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

1. Problem
2. Partition
3. Communicate
4. Agglomerate
5. Map
Graph Embeddings

- Target network may be **virtual network topology**, with nodes usually called **processors** or **processes**.

- 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.

- An alternative methodology is to map tasks and communication onto a graph of the network topology laid out in time, similar to the way we defined butterfly protocols.
Partitioning Strategies

- **Domain partitioning**: subdivide geometric domain into subdomains
- **Functional decomposition**: subdivide algorithm into multiple logical components
- **Independent tasks**: subdivide computation into tasks that do not depend on each other (*embarrassingly parallel*)
- **Array parallelism**: subdivide data stored in vectors, matrices, or other arrays
- **Divide-and-conquer**: subdivide problem recursively into tree-like hierarchy of subproblems
- **Pipelining**: subdivide sequences of tasks performed by the algorithm on each piece of data
Desirable Properties of Partitioning

- Maximum possible *concurrency* in executing resulting tasks, ideally enough to keep all processors busy
- 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
Example: Domain Decomposition

3-D domain partitioned along one (left), two (center), or all three (right) of its dimensions

With 1-D or 2-D partitioning, minimum task size grows with problem size, but not with 3-D partitioning
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
Increasing task sizes can reduce communication but also potentially reduces concurrency

Subtasks that can’t be executed concurrently anyway are obvious candidates for combining into single task

Maintaining balanced workload still important

Replicating computation can eliminate communication and is advantageous if result is cheaper to compute than to communicate
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 \( r \)

\[ \bar{u}_i = \frac{(u_{i-1} + u_{i+1})}{2} \] { update local values }

end

wait for sends to complete

\( u = \bar{u} \)

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

end

$u = \bar{u}$
Surface-to-Volume Effect

- For domain decomposition,
  - *computation* is proportional to *volume* of subdomain
  - *communication* is (roughly) proportional to *surface area* of subdomain

- Higher-dimensional decompositions have more favorable surface-to-volume ratio

- Partitioning across more dimensions yields more neighboring subdomains but smaller total volume of communication than partitioning across fewer dimensions
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
Two communicating tasks can be assigned to

- one processor, avoiding interprocessor communication but sacrificing concurrency
- two adjacent processors, so communication between the tasks is directly supported, or
- two nonadjacent processors, so message routing is required

In general, finding optimal solution to these tradeoffs is NP-complete, so heuristics are used to find effective compromise
For many problems, task graph has regular structure that can make mapping easier.

If communication is mainly global, then communication performance may not be sensitive to placement of tasks on processors, so random mapping may be as good as any.

Random mappings sometimes used deliberately to avoid communication *hot spots*, where some communication links are oversubscribed with message traffic.
Mapping Strategies

With $n$ tasks and $p$ 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

- **block**
  - 0 1 2 3
  - 4 5 6 7
  - 8 9 10 11
  - 12 13 14 15

- **cyclic**
  - 0 4 8 12
  - 1 5 9 13
  - 2 6 10 14
  - 3 7 11 15

- **reflection**
  - 0 7 8 15
  - 1 6 9 14
  - 2 5 10 13
  - 3 4 11 12

- **block-cyclic**
  - 0 1 8 9
  - 2 3 10 11
  - 4 5 12 13
  - 6 7 14 15
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, or use *work stealing* by taking a task assigned to the task-queue of another processor.
- For distributed-memory, the manager/worker paradigm can implement a common task pool, with manager dispatching tasks to workers.
- Better scalability is achieved dynamic load balancing strategies in distributed-memory by periodic global or hierarchical rebalancing.
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.
Example: Atmospheric Flow Model

- Fluid dynamics of atmosphere modeled by system of partial differential equations
- 3-D problem domain discretized by $n_x \times n_y \times n_z$ mesh of points
- Vertical dimension (altitude) $z$, much smaller than horizontal dimensions (latitude and longitude) $x$ and $y$, so $n_z \ll n_x, n_y$
- Derivatives in PDEs approximated by finite differences
- Simulation proceeds through successive discrete steps in time
Example: Atmospheric Flow Model

**Partition:**
- Each fine-grain task computes and stores data values (pressure, temperature, etc) for one mesh point.
- Large-scale problems may require millions or billions of mesh points / tasks.

**Communicate:**
- Finite difference computations at each mesh point use 9-point horizontal stencil and 3-point vertical stencil.
- Solar radiation computations require communication throughout each vertical column of mesh points.
- Global communication to compute total mass of air over domain.
Example: Atmospheric Flow Model

**Agglomerate**:  
- Combine horizontal mesh points in $b \times b$ blocks into coarse-grain tasks to reduce communication for finite differences to exchanges between adjacent nodes.  
- Combine each vertical column of mesh points into single task to eliminate communication for solar computations.  
- Yields $n_x/b \times n_y/b$ coarse-grain tasks.

**Map**:  
- Cyclic or random mapping reduces load imbalance due to solar computations.
Horizontal finite difference stencil for typical point (shaded black) in mesh for atmospheric flow model before (left) and after (right) agglomeration with $b = 2$
References


I. T. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995


T. G. Mattson and B. A. Sanders and B. L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005