Parallel algorithm design
Overview

- Chapter 3 from *Michael J. Quinn*, *Parallel Programming in C with MPI and OpenMP*

- Another resource:

- How to develop a parallel algorithm?
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping
General remarks

- Warning: Parallel algorithm design is not always easily reduced to simple recipes
- May still benefit from a methodical approach
- Intuition for good designs need to be developed gradually
The task/channel model

Directed-graph representation of a parallel computation:
A set of interacting tasks that send/receive messages through channels
A task is a program with local memory and I/O ports
A channel is a message queue
Foster’s design methodology

A four-step process for designing parallel algorithms
I. Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995
Step 1: Partitioning

At the beginning of design, discover as much parallelism as possible

Partitioning: divide the computation and data into pieces

Data-centric partitioning (domain decomposition)
- Data is first divided into pieces, then computation is assigned to data
- Result: a number of tasks, each having some data and a set of operations on the data
  - If an operation requires data from other tasks → communication needed

Computation-centric partitioning (functional decomposition)
- Computation is divided into disjoint tasks, then data is associated with the individual tasks
  - If data is shared between tasks → communication needed
  - Often yield concurrency through pipelining
3 examples of domain decomposition

1-D

2-D

3-D
Example of functional decomposition
Partitioning design checklist

Whichever decomposition, the partitioning result is a collection of *primitive tasks*. The number of tasks is an upper bound on the parallelism we can exploit.

- There are many more tasks than processors
- Redundant computation and data storage are minimized
- Primitive tasks are roughly the same size
- The number of tasks is an increasing function of the problem size
Step 2: Communication

When primitive tasks are identified, determine the communication pattern between them.

There are two types of communication:

- **Local communication**: A task needs values from a small number of neighboring tasks.
- **Global communication**: A significant number of tasks must contribute data to perform a computation.

Communication among tasks is part of parallelization overhead because sequential algorithms never need communication.
2 examples of communication
Communication structure checklist

- The communication operations are balanced among the tasks
- Each task communicates with only a small number of neighbors
- Tasks can perform their communications concurrently
- Tasks can perform their computations concurrently
Step 3: Agglomeration

Now we have a target parallel architecture in mind

Motivation: If the number of tasks exceeds the processors by several orders of magnitude, creating these tasks will be a source of significant overhead. Also non-trivial: “map which tasks to which processors?”

Agglomeration is the process of grouping tasks into larger tasks

The purpose is to improve performance and simplify programming

Typically in MPI programs, one consolidated task per processor (core)

Sometimes, more consolidated tasks than the number of processors
Goal 1: lower communication overhead

(a) Elimination of communication (*increasing the locality*)
(b) Decrease of message transmissions (*saving startup cost—latency*)
Goal 2: maintain scalability

- We don’t want “over-agglomeration”,
  - too few tasks ⇒ maybe not suitable for a future machine with more processors

- Example: a grid of $8 \times 128 \times 256$

- If the second and third dimensions are agglomerated ⇒ 8 tasks
  - Ok for a 4-CPU machine, each CPU responsible for a $2 \times 128 \times 256$ subgrid
  - However, not possible to use more than 8 CPUs
Goal 3: reduce software engineering cost

Suppose a sequential program needs to parallelized.
We should look for an agglomeration that allows us to make great reuse of the existing sequential code.
Time and expense of code development may thus be reduced.
Agglomeration quality checklist

- Locality is increased
- Replicated computations take less time than the communications they replace
- The amount of replicated data is small enough to allow the algorithm to scale
- Agglomerated tasks have similar computational and communication costs
- The number of tasks is an increasing function of the problem size
- The number of tasks is as small as possible, yet at least as great as the number of processors
- The trade-off between the chosen agglomeration and the cost of modification to existing sequential code is reasonable
Step 4: Mapping

- Mapping: assigning tasks to processors
- We focus on distributed-memory architecture
- Goals of mapping: maximize processor utilization and minimize inter-processor communication
- Processor utilization is maximized when the computation is balanced evenly
- Inter-processor communication decreases when two tasks connected by a channel are mapped to the same processor
An example of good mapping
An example of bad mapping

- Left and right processors have 2 tasks, middle processor has 4 tasks.
- Middle processor also has too much communication overhead.
Mapping can be hard

- Increasing processor utilization and minimizing inter-processor communication are often conflicting goals.
  - Suppose $p$ processors are available. Mapping all tasks to a single processor eliminates communication completely, but reduces utilization to $1/p$.

- Also, finding an ideal mapping can be NP-hard.
  - That is, no known polynomial-time algorithm.

- We must rely on heuristics that can do a reasonably good job of mapping.
Typical scenarios

- Domain (data) decomposition $\Rightarrow$ tasks have similar size. If communication pattern is regular, a good strategy is to create $p$ agglomerated tasks for $p$ processors.

- Suppose the number of tasks is fixed and communication pattern is regular, but time to perform each task differs. If nearby tasks have similar computational requirements, then a simple cyclic mapping can give balanced computational load distribution, at the expense of higher communications costs.

- For problems having an unstructured communication pattern, it’s important to minimize the communication overhead. A static load-balancing algorithm can achieve this.

- In case tasks are created and destroyed at runtime, dynamic load-balancing algorithms are needed.
Mapping design checklist

- Designs based on one task per processor and multiple tasks per processor have been considered.
- Both static and dynamic allocations of tasks to processors have been evaluated.
- If a dynamic allocation has been chosen, the manager is not a performance bottleneck.
Example: boundary value problem

Figure 3.8  A thin rod (dark gray) is suspended between two ice baths. The ends of the rod are in contact with the icewater. The rod is surrounded by a thick blanket of insulation. We can use a partial differential equation to model the temperature at any point on the rod as a function of time.
Boundary value problem (2)

- 1D problem in the spatial direction
- Time-dependent problem
- We can use a finite difference mesh: uniform in both spatial and temporal directions
- Let $u_{i,j}$ denote the solution on spatial point $i$ and time level $j$
- Computation formula:

$$ u_{i,j+1} = r u_{i-1,j} + (1 - 2r) u_{i,j} + r u_{i+1,j} $$
Figure 3.10  Data structure used in a finite difference approximation to the rod-cooling problem presented in Figure 3.8. Every point $u_{i,j}$ represents a matrix element containing the temperature at position $i$ on the rod at time $j$. 
Boundary value problem (4)
Example: parallel reduction

- Given a set of \( n \) values: \( a_0, a_1, a_2, \ldots, a_{n-1} \)
- Given an associative binary operator \( \oplus \)
- **Reduction**: compute \( a_0 \oplus a_1 \oplus a_2 \cdots \oplus a_{n-1} \)
- On a sequential computer: \( n - 1 \) operations are needed
- How to implement a parallel reduction?
Parallel reduction (2)
Example: finding global sum
Finding global sum (2)
Another example of parallel reduction

When the number of tasks is not a power of 2
Cost analysis of parallel reduction

- $\chi$: time needed to perform the binary operation $\oplus$
- $\lambda$: time needed to communicate a value from one task to another
- $n$: number of values
- $p$: number of processors
- If $n$ is divided as evenly as possible among $p$ processors
  - Time needed by each processor to treat its assigned values
    \[(\lceil n/p \rceil - 1) \chi\]
  - $\lceil \log p \rceil$ communication steps are needed
  - Each communication step requires time $\lambda + \chi$
  - Total parallel computing time
    \[(\lceil n/p \rceil - 1) \chi + \lceil \log p \rceil (\lambda + \chi)\]
Concluding remarks

- The task/channel model encourages parallel algorithm designs that maximize local computations and minimize communications.

- The algorithm designer typically partitions the computation, identifies communications among primitive tasks, agglomerates primitive tasks into larger tasks, and decides how to map tasks to processors.

- The goals are to maximize processor utilization and minimize inter-processor communications.

- Good designs must often strike a balance between the above two goals.
Real-life example of parallel processing
Exercises

- Exercise 3.2
- Exercise 3.6
- Exercise 3.15
- Exercise 3.16
- Exercise 3.7