Parallel Programming
in C with MPI and OpenMP

Michael J. Quinn
Chapter 3

Parallel Algorithm Design
Outline

• Task/channel model
• Algorithm design methodology
• Case studies
Task/Channel Model

• Parallel computation = set of tasks
• Task
  – Program
  – Local memory
  – Collection of I/O ports
• Tasks interact by sending messages through channels
Task/Channel Model

(a) Memory

(b) Task Channel
Foster’s Design Methodology

• Partitioning
• Communication
• Agglomeration
• Mapping
• **Partitioning**  
  – Computation/data are decomposed.  
  – No practical issues

• **Communication**  
  – Communication structure is determined.

• **Agglomeration**  
  – Evaluate the first two outcomes (performance-wise)  
  – Tasks may get combined.

• **Mapping**  
  – Assign tasks to processors to  
    • Maximize processor utilization  
    • Minimize communication costs  
  – Static/dynamic(runtime)
Foster’s Methodology

Diagram:
- Problem
- Partitioning
- Communication
- Mapping
- Agglomeration
Partitioning

- Expose opportunities for parallel execution.
  - By dividing computation and data into pieces
- As fine-grained as possible.
  - Could be combined later.
- Works on both computation and data.
- Domain decomposition:
  - Focus on data first, then associate computation with data.
- Functional decomposition
  - Decompose computation first, then associate data with the computation.
Domain Decomposition

- Divide the data into small pieces of approximately equal size.
- Determine how to associate computations with the data.
- Partition the computation to be performed.
- Focus on largest data structure first.
- Widely used on spatial data structure
  - 1-D, 2-D, 3-D decomposition
Example Domain Decompositions
Functional Decomposition

- Divide the computation into disjoint tasks.
- Determine how to associate data with the computation
- Examine the data requirements of these tasks
  - If these data requirements are disjoint -> O.K.
Example Functional Decomposition
(interactive image-guided brain surgery)

Acquire Patient Images → Register Images → Track Position of Instruments → Determine Image Locations → Display Image
Partitioning Checklist

- At least 10x more primitive tasks than processors in target computer
- Minimize redundant computations and redundant data storage
- Primitive tasks roughly the same size
- Number of tasks an increasing function of problem size
- Consider several alternatives
Communication

• Determine values passed among tasks
• Easier for functional decomposition
  – Corresponds to data flow between tasks.
• Types
  – Local/global communication
  – Structured/unstructured
  – Static/dynamic
  – Synchronous/asynchronous
Communication

• Determine values passed among tasks
• Easier for functional decomposition
  – Corresponds to data flow between tasks.
• Types
  – Local/global communication
  – Structured/unstructured
  – Static/dynamic
  – Synchronous/asynchronous
Communication

• Determine values passed among tasks
• Easier for functional decomposition
  – Corresponds to data flow between tasks.
• Types
  – Local/global communication
  – Structured/unstructured
  – Static/dynamic
  – Synchronous/asynchronous
Communication

• Local communication
  – Task needs values from a small number of other tasks
  – Create channels illustrating data flow

• Global communication
  – Significant number of tasks contribute data to perform a computation
  – Don’t create channels for them early in design
Communication Checklist

- Communication operations balanced among tasks
- Each task communicates with only small group of neighbors
- Tasks can perform communications concurrently
- Task can perform computations concurrently
Agglomeration (abstract -> concrete)

- The result so far is abstract that it’s not specialized for efficient execution on any particular computer.
- Would it be useful to combine tasks, replicate data and/or computation?
- Sometimes conflicting goals
  - Reducing communication costs by increasing computation and communication granularity
  - Retaining flexibility w.r.t scalability and mapping decisions
  - Reducing Software engineering costs
Agglomeration

• Grouping tasks into larger tasks

• Goals
  – Improve performance
  – Maintain scalability of program
  – Simplify programming

• In MPI programming, goal often to create one agglomerated task per processor
Agglomeration (abstract -> concrete)

- Reducing communication costs by increasing computation and communication granularity
  - Sending less data, fewer messages, create fewer tasks
  - Surface to volume effects
    - Large granule -> comm/computation ration small
  - Replicating computation
    - Summation and broadcast => summation by all
- Avoiding communication
  - Combining tasks at different level which can’t execute concurrently
Agglomeration (abstract $\rightarrow$ concrete)

- Retaining flexibility w.r.t scalability and mapping decisions
  - The ability to create a varying number of tasks is critical to be portable/scalable
  - Overlapping communication/computation
  - Balanced load over processors

- Reducing Software engineering costs
  - Not only to improve efficiency and flexibility, it is important to reduce development costs with parallelizing.
Agglomeration Can Improve Performance

- Eliminate communication between primitive tasks agglomerated into consolidated task
- Combine groups of sending and receiving tasks
Agglomeration Checklist

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

• 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
  – Minimize interprocessor communication
Mapping Example

(a)

(b)
Optimal Mapping

- Finding optimal mapping is NP-hard
- Must rely on heuristics
How to map?

Goal: minimize total execution time

Domain decomposition with fixed # of equal sized tasks and structural communication
  – Efficient mapping is straight forward. (minimize IPC)

• Complex algorithms with variable amount of work per tasks
  – Use load balancing.

• Dynamic # of tasks/communication
  – Dynamic load balancing.
Load Balancing Algorithms (1)

- Recursive bisection (divide and conquer)
  - Recursive coordinate bisection:
    - Irregular grids with mostly local communication.
  - Unbalanced recursive bisection:
    - Optimize communication with better aspect ratio.
  - Recursive graph bisection:
    - Use connectivity info to reduce communication
  - Recursive spectral bisection
Load Balancing Algorithms (2)

- Local algorithms can be used dynamically
  - Inexpensive since no need of global info
- Probabilistic methods
  - Adv: low cost and scalable
  - Dis: off processor communication is needed
- Cyclic mapping
  - Computation load per grid point varies and significant spatial locality
  - Load balanced, but communication cost increases
Task scheduling Algorithms

- Manager/worker
- Hierarchical manager/worker
- Decentralized
  - Separate task pool is maintained on each processor,
  - Idle workers request problems from other processors.
- Termination detection (when to stop requesting?)
Mapping Decision Tree

• Static number of tasks
  – Structured communication
    • Constant computation time per task
      – Agglomerate tasks to minimize comm
      – 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
Mapping Strategy

- Static number of tasks
- Dynamic number of tasks
  - Frequent communications between tasks
    - Use a dynamic load balancing algorithm
  - Many short-lived tasks
    - Use a run-time task-scheduling algorithm
Mapping Checklist

- Considered designs based on one task per processor and multiple tasks per processor
- Evaluated static and dynamic task allocation
- If dynamic task allocation chosen, task allocator is not a bottleneck to performance
- If static task allocation chosen, ratio of tasks to processors is at least 10:1
Case Studies

- Boundary value problem
- Finding the maximum
- The n-body problem
- Adding data input
Boundary Value Problem

Ice water  Rod  Insulation
Rod Cools as Time Progresses
Finite Difference Approximation

Time

Space

\[ u = 100 \sin(\pi x) \]
Finite Difference Method

• \( h \) : how fine is the mesh \( x \)-wise ?
  – smaller \( h \) \( \rightarrow \) finer
• \( k \) : how fine is the mesh \( y \)-wise ?
  – smaller \( k \) \( \rightarrow \) finer
• \( u_{i,j+1} = r \ u_{i-1,j} + (1-2r)u_{i,j} + r \ u_{i+1,j} \)
• \( r = \frac{k}{h^2} \)
Partitioning

- One data item per grid point
- Associate one primitive task with each grid point
- Two-dimensional domain decomposition
Communication

• Identify communication pattern between primitive tasks
• Each interior primitive task has three incoming and three outgoing channels
Agglomeration and Mapping
Sequential execution time

- $\chi$ – time to update element
- $n$ – number of elements
- $m$ – number of iterations
- Sequential execution time: $m(n-1)\chi$
Parallel Execution Time

- $\rho$ – number of processors
- $\lambda$ – message latency
- Parallel execution time $m(\chi[(n-1)/\rho]+2\lambda)$
Reduction

- Given associative operator $\oplus$
- $a_0 \oplus a_1 \oplus a_2 \oplus ... \oplus a_{n-1}$
- Examples
  - Add
  - Multiply
  - And, Or
  - Maximum, Minimum
Parallel Reduction Evolution

\( n-1 \) tasks
Parallel Reduction Evolution

\[ \frac{n}{2} - 1 \text{ tasks} \]

\[ \frac{n}{2} - 1 \text{ tasks} \]
Parallel Reduction Evolution

\[
\frac{n}{4} - 1 \text{ tasks}
\]

\[
\frac{n}{4} - 1 \text{ tasks}
\]

\[
\frac{n}{4} - 1 \text{ tasks}
\]

\[
\frac{n}{4} - 1 \text{ tasks}
\]
Binomial Trees

Subgraph of hypercube
Finding Global Sum
Finding Global Sum

1 → 7 → -6 → 4

4 → 5 → 8 → 2
Finding Global Sum

\[
\begin{array}{ccc}
8 & & -2 \\
9 & & 10
\end{array}
\]
Finding Global Sum
Finding Global Sum

Binomial Tree

25
Agglomeration
Agglomeration
The n-body Problem
The n-body Problem
Partitioning

- Domain partitioning
- Assume one task per particle
- Task has particle’s position, velocity vector
- Iteration
  - Get positions of all other particles
  - Compute new position, velocity
Gather
All-gather
Complete Graph for All-gather
Hypercube for All-gather
Communication Time

Complete graph

\[(p - 1)(\lambda + \frac{n}{p}) = (p - 1)\lambda + \frac{n(p - 1)}{\beta p}\]

Hypercube

\[\log p \sum_{i=1}^{p} \left( \lambda + \frac{2^{i-1} n}{\beta p} \right) = \lambda \log p + \frac{n(p - 1)}{\beta p}\]
Adding Data Input

- Input
- Output
Scatter
Scatter in log $\rho$ Steps
Summary: Task/channel Model

- Parallel computation
  - Set of tasks
  - Interactions through channels
- Good designs
  - Maximize local computations
  - Minimize communications
  - Scale up
Summary: Design Steps

• Partition computation
• Agglomerate tasks
• Map tasks to processors
• Goals
  – Maximize processor utilization
  – Minimize inter–processor communication
Summary: Fundamental Algorithms

- Reduction
- Gather and scatter
- All-gather
Functional Decomposition

• Divide the computation into disjoint tasks.
• Determine how to associate data with the computation
• Examine the data requirements of these tasks
  – If these data requirements are disjoint → O.K.