Patterns for Parallel Programming

Textbook

First of all

- Is the problem large enough and the results significant enough to justify the effort to solve it faster?

- If so, what are the most computationally intensive parts? Whether speeding them up provides sufficient performance gains (i.e., Amdahl’s law)?

Overview

- Finding Concurrency
  - Exposing exploitable concurrency

- Algorithm Structure
  - Structuring algorithm to take advantage

- Supporting structures
  - Structuring program and shared data

- Implementation mechanisms
  - Mapping to particular programming environment
Finding Concurrency

Decomposition Patterns

- Task decomposition: view problem as a stream of instructions that can be broken into sequences called tasks that can execute in parallel.
  - Key: Independent operations

- Data decomposition: view problem from data perspective and focus on how they can be broken into distinct chunks
  - Key: Data chunks that can be operated upon independently

- Task and data decomposition imply each other. They are different facets of the same fundamental decomposition

University of Central Florida
Example

• Matrix multiplication
  – Task decomposition
    • Considering the computation of each element in the product matrix as a separate task
    • Performs poorly => group tasks pattern
  – Data decomposition
    • Decompose the product matrix into chunks, e.g., one row a chunk, or a small submatrix (or block) per chunk

Dependency analysis patterns

• Group tasks: group tasks that have the same dependency constraints; identify which tasks must execute concurrently
  – Reduced synchronization overhead – all tasks in the group can use a barrier to wait for a common dependence
  – All tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shard Memory)
  – Grouping and merging dependent tasks into one task reduces need for synchronization

• Order task pattern: identifying order constraints among task groups.
  – Control dependency: Find the task group that creates it
  – Data dependency: temporal order for producer and consumer relationship
Dependency analysis patterns

• Data sharing pattern: how data is shared among the tasks?
  – Read only: make own local copies
  – Effectively local: the shared data is partitioned into subsets, each of which is accessed (for read or write) by only one task at a time.
  – Read-write: the data is accessed by more than one task. Need exclusive access mechanisms.
  – Example: the use of the shared memory among threads in a thread block.

Design Evaluation Pattern

• Whether the partition fits the target hardware platform?

• Key questions to ask
  – How many threads can be supported?
  – How many threads are needed?
  – How are the data structures shared?
  – Is there enough work in each thread between synchronizations to make parallel execution worthwhile?
Algorithm Structure

Finding Concurrency

Organize by Tasks
- Task Parallelism
- Divide and Conquer

Organize by Data
- Geometric Decomposition
- Recursive Data Decomposition

Organize by Flow
- Pipeline
- Event Condition

Supporting structures

Implementation mechanisms

Organizing Principle

Start

Organize by Task
- Linear
- Recursive
- Task Parallelism
- Divide and Conquer

Organize by Data
- Linear
- Recursive
- Geometric Decomposition

Organize by Data Flow
- Recursive
- Regular
- Irregular
- Pipeline
- Event Driven
Task Parallelism Pattern

- After the problem is decomposed into a collection of tasks that can execute concurrently, how to exploit this concurrency efficiently?

- Load balancing

Divide and Conquer Pattern

- If the problem is formulated using the sequential divide-and-conquer strategy, how to exploit the potential concurrency?
Divide-and-Conquer Pattern

- Sequential code

```java
class Solution {
    public static Solution solve(Problem P) {
        if (baseCase(P))
            return baseSolution(P);
        else {
            Problem subProblems[N];
            Solution subSolutions[N];
            subProblems = split(P);
            for (int i = 0; i < N; i++)
                subSolutions[i] = solve(subProblems[i]);
            return merge(subSolutions);
        }
    }
}
```

- Parallelization Strategy
**Geometric Decomposition Pattern**

- How to organize the algorithm after the data has been decomposed into concurrently updatable chunks?
- Decomposition to minimize the data communication and dependency among tasks
- Care needs to be taken when update non-local data, e.g., exchange operations

---

**Recursive data pattern**

- Suppose the problem involves an operation on a recursive data structure that appears to require sequential processing. How to make the operations on these data structures parallel?
- Check whether divide-and-conquer pattern works
- If not, may need to transform the original algorithm.
Example: Finding root in a forest

Supporting Structures

Finding Concurrency

Algorithm Structure

Program Structures
- SPMD
- Master/Worker
- Loop Parallelism
- Fork/Join

Data Structures
- Shared Data
- Shared Queue
- Distributed Array

Implementation mechanisms

University of Central Florida
### Relationship between Supporting Program Structure Patterns and Algorithm Structure Patterns

<table>
<thead>
<tr>
<th></th>
<th>Task Parallel</th>
<th>Divide/Conquer</th>
<th>Geometric Decomp.</th>
<th>Geometric Recursive Data</th>
<th>Pipeline</th>
<th>Event-based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPMD</strong></td>
<td>☺☺☺☺</td>
<td>☺☺</td>
<td>☺☺☺</td>
<td>☺☺☺</td>
<td>☺☺☺☺☺</td>
<td>☺☺</td>
</tr>
<tr>
<td><strong>Loop Parallel</strong></td>
<td>☺☺☺☺</td>
<td>☺☺</td>
<td></td>
<td>☺☺</td>
<td>☺☺☺☺☺</td>
<td>☺</td>
</tr>
<tr>
<td><strong>Master/Worker</strong></td>
<td>☺☺☺☺</td>
<td>☺☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td><strong>Fork/Join</strong></td>
<td>☺☺</td>
<td>☺☺☺</td>
<td>☺☺</td>
<td>☺</td>
<td>☺☺</td>
<td>☺☺☺</td>
</tr>
</tbody>
</table>

### Relationship between Supporting Program Structure Patterns and Programming Environment

<table>
<thead>
<tr>
<th></th>
<th>OpenMP</th>
<th>MPI</th>
<th>Java</th>
<th>Brook+/CUDA</th>
<th>Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPMD</strong></td>
<td>☺☺☺</td>
<td>☺☺☺☺☺☺☺☺</td>
<td>☺☺</td>
<td>☺☺☺☺☺☺☺☺☺</td>
<td>☺☺☺☺☺☺</td>
</tr>
<tr>
<td><strong>Loop Parallel</strong></td>
<td>☺☺☺</td>
<td>☺☺</td>
<td>☺☺☺☺☺☺☺☺</td>
<td></td>
<td>☺☺☺☺☺</td>
</tr>
<tr>
<td><strong>Master/Slave</strong></td>
<td>☺☺☺</td>
<td>☺☺☺☺☺☺☺☺</td>
<td>☺☺☺☺☺☺☺☺</td>
<td></td>
<td>☺☺☺☺☺☺</td>
</tr>
<tr>
<td><strong>Fork/Join</strong></td>
<td>☺☺☺</td>
<td>☺☺☺☺☺☺☺☺</td>
<td>☺☺☺☺☺☺☺☺</td>
<td></td>
<td>☺☺☺☺</td>
</tr>
</tbody>
</table>

University of Central Florida
Implementation Mechanisms

- Finding Concurrency
- Algorithm Structure
- Supporting structures
  - Thread/Process management
  - Synchronization
  - Communication