EE382N (20): Computer Architecture - Parallelism and Locality
Fall 2011
Lecture 12 – Parallelism in Software III

Mattan Erez

The University of Texas at Austin
Credits

• Most of the slides courtesy Dr. Rodric Rabbah (IBM)
  – Taken from 6.189 IAP taught at MIT in 2007.
Patterns for Parallelizing Programs

4 Design Spaces

Algorithm Expression

• Finding Concurrency
  – Expose concurrent tasks

• Algorithm Structure
  – Map tasks to processes to exploit parallel architecture

Software Construction

• Supporting Structures
  – Code and data structuring patterns

• Implementation Mechanisms
  – Low level mechanisms used to write parallel programs

Quick recap

• Decomposition
  – Keep things general and simple
    • Consider rough machine properties only (10, 1000, 1M, …)
  – Task
    • Natural in some programs
    • Need to balance overheads of fine-grained with degree of par.
  – Data
    • Natural in some programs, less general than task
    • Consider data structure
  – Pipeline
    • Overlap compute and comm.
    • Reduce the degree of other parallelism needed

• Dependencies
  – Equivalent to RAW/WAW/WAR
Patterns for Parallelizing Programs

4 Design Spaces

Algorithm Expression

- Finding Concurrency
  - Expose concurrent tasks

- Algorithm Structure
  - Map tasks to processes to exploit parallel architecture

Software Construction

- Supporting Structures
  - Code and data structuring patterns

- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs

Algorithm Structure Design Space

• Given a collection of concurrent tasks, what’s the next step?
• Map tasks to units of execution (e.g., threads)

• Important considerations
  – Magnitude of number of execution units platform will support
  – Cost of sharing information among execution units
  – Avoid tendency to over constrain the implementation
    • Work well on the intended platform
    • Flexible enough to easily adapt to different architectures
Major Organizing Principle

• How to determine the algorithm structure that represents the mapping of tasks to units of execution?

• Concurrency usually implies major organizing principle
  – Organize by tasks
  – Organize by data decomposition
  – Organize by flow of data
Organize by Tasks?

Recursive?

- yes → Divide and Conquer
- no → Task Parallelism
Task Parallelism

• Molecular dynamics
  – Non-bonded force calculations, some dependencies

• Common factors
  – Tasks are associated with iterations of a loop
  – Tasks largely known at the start of the computation
  – All tasks may not need to complete to arrive at a solution
Divide and Conquer

• For recursive programs: divide and conquer
  – Subproblems may not be uniform
  – May require dynamic load balancing
Organize by Data?

- Operations on a central data structure
  - Arrays and linear data structures
  - Recursive data structures
Recursive Data

• Computation on a list, tree, or graph
  – Often appears the only way to solve a problem is to sequentially move through the data structure

• There are however opportunities to reshape the operations in a way that exposes concurrency
Recursive Data Example: Find the Root

- Given a forest of rooted directed trees, for each node, find the root of the tree containing the node
  - Parallel approach: for each node, find its successor’s successor, repeat until no changes
  - $O(\log n)$ vs. $O(n)$
Work vs. Concurrency Tradeoff

• Parallel restructuring of find the root algorithm leads to $O(n \log n)$ work vs. $O(n)$ with sequential approach

• Most strategies based on this pattern similarly trade off increase in total work for decrease in execution time due to concurrency
Organize by Flow of Data?

- In some application domains, the flow of data imposes ordering on the tasks
  - Regular, one-way, mostly stable data flow
  - Irregular, dynamic, or unpredictable data flow

```
Regular?

yes

Pipeline

no

Event-based Coordination
```
Pipeline Throughput vs. Latency

- Amount of concurrency in a pipeline is limited by the number of stages

- Works best if the time to fill and drain the pipeline is small compared to overall running time

- Performance metric is usually the throughput
  - Rate at which data appear at the end of the pipeline per time unit (e.g., frames per second)

- Pipeline latency is important for real-time applications
  - Time interval from data input to pipeline, to data output
Event-Based Coordination

- In this pattern, interaction of tasks to process data can vary over unpredictable intervals

- Deadlocks are a danger for applications that use this pattern
  - Dynamic scheduling has overhead and may be inefficient
    - Granularity a major concern

- Another option is various “static” dataflow models
  - E.g., synchronous dataflow
Patterns for Parallelizing Programs

4 Design Spaces

Algorithm Expression

• Finding Concurrency
  – Expose concurrent tasks

• Algorithm Structure
  – Map tasks to processes to exploit parallel architecture

Software Construction

• Supporting Structures
  – Code and data structuring patterns

• Implementation Mechanisms
  – Low level mechanisms used to write parallel programs

Code Supporting Structures

- Loop parallelism
- Master/Worker
- Fork/Join
- SPMD
- **Map/Reduce**
Loop Parallelism Pattern

• Many programs are expressed using iterative constructs
  – Programming models like OpenMP provide directives to automatically assign loop iteration to execution units
  – Especially good when code cannot be massively restructured

```c
#pragma omp parallel for
for(i = 0; i < 12; i++)
    C[i] = A[i] + B[i];
```
Master/Worker Pattern

Independent Tasks

master

worker

worker

worker

worker

worker
Master/Worker Pattern

• Particularly relevant for problems using task parallelism pattern where tasks have no dependencies
  – Embarrassingly parallel problems

• Main challenge in determining when the entire problem is complete
Fork/Join Pattern

• Tasks are created dynamically
  – Tasks can create more tasks

• Manages tasks according to their relationship

• Parent task creates new tasks (fork) then waits until they complete (join) before continuing on with the computation
SPMD Pattern

- Single Program Multiple Data: create a single source-code image that runs on each processor
  - Initialize
  - Obtain a unique identifier
  - Run the same program each processor
    - Identifier and input data differentiate behavior
  - Distribute data
  - Finalize
SPMD Challenges

• Split data correctly

• Correctly combine the results

• Achieve an even distribution of the work

• For programs that need dynamic load balancing, an alternative pattern is more suitable
Map/Reduce Pattern

• Two phases in the program
• Map phase applies a single function to all data
  – Each result is a tuple of value and tag
• Reduce phase combines the results
  – The values of elements with the same tag are combined to a single value per tag -- reduction
  – Semantics of combining function are associative
  – Can be done in parallel
  – Can be pipelined with map
• Google uses this for all their parallel programs
Communication and Synchronization Patterns

• Communication
  – Point-to-point
  – Broadcast
  – Reduction
  – Multicast

• Synchronization
  – Locks (mutual exclusion)
  – Monitors (events)
  – Barriers (wait for all)
    • Split-phase barriers (separate signal and wait)
      – Sometimes called “fuzzy barriers”
    • Named barriers allow waiting on subset
## Algorithm Structure and Organization (from the Book)

<table>
<thead>
<tr>
<th></th>
<th>Task parallelism</th>
<th>Divide and conquer</th>
<th>Geometric decomposition</th>
<th>Recursive data</th>
<th>Pipeline</th>
<th>Event-based coordination</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 Parallelism</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>

- Patterns can be hierarchically composed so that a program uses more than one pattern