EE382N (20): Computer Architecture - Parallelism and Locality
Spring 2015
Lecture 15 – 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

• Parallel Scan slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  – Taken from EE493-AI taught at UIUC in Sprig 2007
4 Common Steps to Creating a Parallel Program

Partitioning

Sequential Computation → Tasks → Units of Execution → Parallel Program → Processors

Decomposition → Assignment → Orchestration → Mapping
Data Decomposition Examples

- Molecular dynamics
  - Geometric decomposition

- Merge sort
  - Recursive decomposition
Dependence Analysis

- Given two tasks how to determine if they can safely run in parallel?
Bernstein’s Condition

• \( R_i \): set of memory locations read (input) by task \( T_i \)
• \( W_j \): set of memory locations written (output) by task \( T_j \)

• Two tasks \( T_1 \) and \( T_2 \) are parallel if
  – input to \( T_1 \) is not part of output from \( T_2 \)
  – input to \( T_2 \) is not part of output from \( T_1 \)
  – outputs from \( T_1 \) and \( T_2 \) do not overlap
Example

\[ T_1 \]
\[ a = x + y \]

\[ T_2 \]
\[ b = x + z \]

\[ R_1 = \{ x, y \} \]
\[ W_1 = \{ a \} \]

\[ R_2 = \{ x, z \} \]
\[ W_2 = \{ b \} \]

\[ R_1 \cap W_2 = \phi \]
\[ R_2 \cap W_1 = \phi \]
\[ W_1 \cap W_2 = \phi \]
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?
```

- yes: Recursive Data
- no: Geometric Decomposition
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

[Diagram:
  - Regular? (yes -> Pipeline)
  - Event-based Coordination (no)]
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
- \textit{Map}/\textit{Reduce}
- \textit{Task} dataflow
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

A
B
C
D
E

A
B
C
D
E

worker
worker
worker
worker
Master/Worker Pattern

• Particularly relevant for problems using task parallelism pattern where task 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
Quick recap

• Decomposition
  – High-level and fairly abstract
  – Consider machine scale for the most part
  – Task, Data, Pipeline
  – Find dependencies

• Algorithm structure
  – Still abstract, but a bit less so
  – Consider communication, sync, and bookkeeping
  – Task (collection/recursive)
  – Data (geometric/recursive)
  – Dataflow (pipeline/event-based-coordination)

• Supporting structures
  – Loop
  – Master/worker
  – Fork/join
  – SPMD
  – MapReduce
## 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
Algorithm Structure and Organization (my view)

<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>SPMD</td>
<td>****</td>
<td>**</td>
<td>****</td>
<td>**</td>
<td>*****</td>
<td>*</td>
</tr>
<tr>
<td>Loop Parallelism</td>
<td>**** when no dependencies</td>
<td>*</td>
<td>****</td>
<td>*</td>
<td>*****</td>
<td>SWP to hide comm.</td>
</tr>
<tr>
<td>Master/Worker</td>
<td>****</td>
<td>***</td>
<td>***</td>
<td>***</td>
<td>**</td>
<td>*****</td>
</tr>
<tr>
<td>Fork/Join</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
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

ILP, DLP, and TLP in SW and HW

• ILP
  - OOO
  - Dataflow
  - VLIW

• DLP
  - SIMD
  - Vector

• TLP
  - Essentially multiple cores with multiple sequencers

• ILP
  - Within straight-line code

• DLP
  - Parallel loops
  - Tasks operating on disjoint data
    - No dependencies within parallelism phase

• TLP
  - All of DLP +
  - Producer-consumer chains
# ILP, DLP, and TLP and Supporting Patterns

<table>
<thead>
<tr>
<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>ILP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DLP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TLP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## ILP, DLP, and TLP and Supporting Patterns

<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>ILP</strong></td>
<td>inline / unroll</td>
<td>inline</td>
<td>unroll</td>
<td>inline</td>
<td>inline / unroll</td>
<td>inline</td>
</tr>
<tr>
<td><strong>DLP</strong></td>
<td>natural or local-conditions</td>
<td>after enough divisions</td>
<td>natural</td>
<td>after enough branches</td>
<td>difficult</td>
<td>local-conditions</td>
</tr>
<tr>
<td><strong>TLP</strong></td>
<td>natural</td>
<td>natural</td>
<td>natural</td>
<td>natural</td>
<td>natural</td>
<td>natural</td>
</tr>
</tbody>
</table>

**ILP** (Instruction Level Parallelism): Inline or unroll, divide and conquer, geometric decomposition, inline recursive data, pipeline inline / unroll, event-based coordination inline.

**DLP** (Data Level Parallelism): Natural or local conditions, after enough divisions, natural geometric decomposition, after enough branches, difficult pipeline, local conditions.

**TLP** (Task Level Parallelism): Natural task parallelism, natural divide and conquer, natural geometric decomposition, natural recursive data, natural pipeline, natural event-based coordination. 
## ILP, DLP, and TLP and Implementation Patterns

<table>
<thead>
<tr>
<th></th>
<th>SPMD</th>
<th>Loop Parallelism</th>
<th>Mater/Worker</th>
<th>Fork/Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## ILP, DLP, and TLP and Implementation Patterns

<table>
<thead>
<tr>
<th></th>
<th>SPMD</th>
<th>Loop Parallelism</th>
<th>Master/Worker</th>
<th>Fork/Join</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ILP</strong></td>
<td>pipeline</td>
<td>unroll</td>
<td>inline</td>
<td>inline</td>
</tr>
<tr>
<td><strong>DLP</strong></td>
<td>natural or local-conditional</td>
<td>natural</td>
<td>local-conditional</td>
<td>after enough divisions + local-conditional</td>
</tr>
<tr>
<td><strong>TLP</strong></td>
<td>natural</td>
<td>natural</td>
<td>natural</td>
<td>natural</td>
</tr>
</tbody>
</table>
Outline

• Molecular dynamics example
  – Problem description
  – Steps to solution
    • Build data structures; Compute forces; Integrate for new positions; Check global solution; Repeat
  – Finding concurrency
    • Scans; data decomposition; reductions
  – Algorithm structure
  – Supporting structures
GROMACS

- Highly optimized molecular-dynamics package
  - Popular code
  - Specifically tuned for protein folding
  - Hand optimized loops for SSE3 (and other extensions)
Gromacs Components

- Non-bonded forces
  - Water-water with cutoff
  - Protein-protein tabulated
  - Water-water tabulated
  - Protein-water tabulated

- Bonded forces
  - Angles
  - Dihedrals

- Boundary conditions

- Verlet integrator

- Constraints
  - SHAKE
  - SETTLE

- Other
  - Temperature-pressure coupling
  - Virial calculation
GROMACS Water-Water Force Calculation

- Non-bonded long-range interactions
  - Coulomb
  - Lennard-Jones
  - 234 operations per interaction

\[ V_{nb} = \sum_{i,j} \left[ \frac{1}{4\pi\varepsilon_0} \frac{q_i q_j}{r_{ij}} + \left( \frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^6} \right) \right] \]

Water-water interaction ~75% of GROMACS run-time
GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is $o(n^2)$
- GROMACS approximates with a cutoff
  - Molecules located more than $r_c$ apart do not interact
  - $O(nr_c^3)$

Efficient algorithm leads to variable rate input streams
GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is $O(n^2)$
- GROMACS approximates with a cutoff
  - Molecules located more than $r_c$ apart do not interact
  - $O(nr_c^3)$

Efficient algorithm leads to variable rate input streams
GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is $o(n^2)$
- GROMACS approximates with a cutoff
  - Molecules located more than $r_c$ apart do not interact
  - $O(nr_c^3)$

Efficient algorithm leads to variable rate input streams
GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is $o(n^2)$
- GROMACS approximates with a cutoff
  - Molecules located more than $r_c$ apart do not interact
  - $O(nr_c^3)$

Efficient algorithm leads to variable rate input streams
GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is \( o(n^2) \)
- GROMACS approximates with a cutoff
  - Molecules located more than \( r_c \) apart do not interact
  - \( O(nr_c^3) \)
- Separate neighbor-list for each molecule
  - Neighbor-lists have variable number of elements

Efficient algorithm leads to variable rate input streams
Other Examples

• More patterns
  – Reductions
  – Scans
    • Building a data structure

• More examples
  – Search
  – Sort
  – FFT as divide and conquer
  – Structured meshes and grids
  – Sparse algebra
  – Unstructured meshes and graphs
  – Trees
  – Collections
    • Particles
    • Rays