Designing for Parallelism
4 Common Steps to Creating a Parallel Program

Partitioning

Sequential computation → Tasks → Processes → Parallel program → Processors

Decomposition → Assignment → Orchestration → Mapping
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

\[ a = x + y \]

\[ b = x + z \]

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

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

\[ R_1 \cap W_2 = \emptyset \]
\[ R_2 \cap W_1 = \emptyset \]
\[ W_1 \cap W_2 = \emptyset \]
Decomposition (Amdahl’s Law)

- Identify concurrency and decide at what level to exploit it

- Break up computation into tasks to be divided among processes
  - Tasks may become available dynamically
  - Number of tasks may vary with time

- Enough tasks to keep processors busy
  - Number of tasks available at a time is upper bound on achievable speedup
Limits to Performance Scalability

- Programs have sequential parts and parallel parts

Sequential part (data dependence)

Parallel part (no data dependence)

```
a = b + c;
d = a + 1;
e = d + a;
for (i=0; i < e; i++)
    M[i] = 1;
```
Amdahl’s Law

- Potential program speedup is defined by the fraction of code that can be parallelized.

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 seconds</td>
<td>50 seconds</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>25 seconds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25 seconds</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>10 seconds</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>25 seconds</td>
</tr>
<tr>
<td>100 seconds</td>
<td>60 seconds</td>
</tr>
</tbody>
</table>

Use 5 processors for parallel work:

<table>
<thead>
<tr>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 seconds</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>10 seconds</td>
</tr>
<tr>
<td>+</td>
</tr>
<tr>
<td>25 seconds</td>
</tr>
<tr>
<td>60 seconds</td>
</tr>
</tbody>
</table>
Amdahl’s Law

- **Speedup** = \( \frac{\text{old running time}}{\text{new running time}} \)
- \( = \frac{100 \text{ seconds}}{60 \text{ seconds}} \)
- \( = 1.67 \)
- (parallel version is 1.67 times faster)
Amdahl’s Law

- $p =$ fraction of work that can be parallelized
- $n =$ the number of processor

$$speedup = \frac{\text{old running time}}{\text{new running time}}$$

$$= \frac{1}{(1-p) + \frac{p}{n}}$$

- fraction of time to complete sequential work
- fraction of time to complete parallel work
Implications of Amdahl’s Law

- Speedup tends to $\frac{1}{1-p}$ as number of processors tends to infinity

- Parallel programming is worthwhile when programs have a lot of work that is parallel in nature
4 Common Steps to Creating a Parallel Program

- Sequential computation
- Tasks
- Processes
- Parallel program
- Processors

Partitioning
Assignment (Granularity)

- Specify mechanism to divide work among core
  - Balance work and reduce communication

- Structured approaches usually work well
  - Code inspection or understanding of application
  - Well-known design patterns

- As programmers, we worry about partitioning first
  - Independent of architecture or programming model
  - But complexity often affect decisions!
Granularity

- Granularity is a qualitative measure of the ratio of computation to communication.

- Computation stages are typically separated from periods of communication by synchronization events.
Fine vs. Coarse Granularity

- Fine-grain Parallelism
  - Low computation to communication ratio
  - Small amounts of computational work between communication stages
  - Less opportunity for performance enhancement
  - High communication overhead

- Coarse-grain Parallelism
  - High computation to communication ratio
  - Large amounts of computational work between communication events
  - More opportunity for performance increase
  - Harder to load balance efficiently
The Load Balancing Problem

- Processors that finish early have to wait for the processor with the largest amount of work to complete
  - Leads to idle time, lowers utilization

```c
// start parallel computation
for (int i = 0; i < n; i++) {
    start work on processor i
}

// waits for completion message
for (int i = 0; i < n; i++) {
    while (processor i not done);
}
```
Static Load Balancing

- Programmer make decisions and assigns a fixed amount of work to each processing core a priori.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Minimal runtime overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Complex static analysis possible</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disadvantages</th>
<th>Requires good static work estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No runtime adaptation for work variance</td>
</tr>
</tbody>
</table>

P1 P2

work queue
Dynamic Load Balancing

- When one core finishes its allocated work, it takes on work from core with the heaviest workload.
- Ideal for codes where work is uneven, and in heterogeneous multicore.
Dynamic Load Balancing

- When one core finishes its allocated work, it takes on work from core with the heaviest workload
- Ideal for codes where work is uneven, and in heterogeneous multicore

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjusts runtime distribution of work among the cores</td>
<td>Additional communication and scheduling overhead</td>
</tr>
<tr>
<td>Requires less work from the programmer and compiler</td>
<td></td>
</tr>
</tbody>
</table>
Granularity and Performance Tradeoffs

1. Load balancing
   - How well is work distributed among cores?

2. Synchronization
   - Are there ordering constraints on execution?

3. Communication
   - Communication is not cheap!
Communication Cost Model

\[ C = f \times (o + l + \frac{n}{m} + t - overlap) \]

- total data sent
- number of messages
- frequency of messages
- overhead per message (at both ends)
- network delay per message
- bandwidth along path (determined by network)
- cost induced by contention per message
- amount of latency hidden by concurrency with computation
Types of Communication

- cores exchange data or control messages
- control messages are often short
- data messages are relatively much larger
- is different interconnect needed for control and data messages?
4 Common Steps to Creating a Parallel Program

1. Sequential computation
2. Decomposition
3. Assignment
4. Orchestration
5. Mapping

Partitioning

- Tasks
- Processes
- Parallel program
- Processors
Orchestration and Mapping (Locality)

- Computation and communication concurrency
- **Preserve locality of data**
- Schedule tasks to satisfy dependences early
Locality of Memory Accesses (Shared Memory)

for (i = 0; i < 16; i++)
    C[i] = A[i] + ...;

fork (threads)

join (barrier)
Locality of Memory Accesses (Shared Memory)

```
for (i = 0; i < 16; i++)
    C[i] = A[i] + ...;
```

- Parallel computation is serialized due to memory contention and lack of bandwidth
Locality of Memory Accesses
(Shared Memory)

for (i = 0; i < 16; i++)
    C[i] = A[i] + ...;

- Distribute data to relieve contention and increase effective bandwidth
Parallel Programming Using Patterns
Parallel Programming by Pattern

- Provides a cookbook to systematically guide programmers
  - Decompose, Assign, Orchestrate, Map
  - Can lead to high quality solutions in some domains

- Provide common vocabulary to the programming community
  - Each pattern has a name, providing a vocabulary for discussing solutions

- Helps with software reusability, malleability, and modularity
  - Written in prescribed format to allow the reader to quickly understand the solution and its context

- Otherwise, too difficult for programmers, and software will not fully exploit parallel hardware
History

- Berkeley architecture professor Christopher Alexander

- In 1977, patterns for city planning, landscaping, and architecture in an attempt to capture principles for “living” design
Therefore:

Whenever you build a balcony, a porch, a gallery, or a terrace always make it at least six feet deep. If possible, recess at least a part of it into the building so that it is not cantilevered out and separated from the building by a simple line, and enclose it partially.
Patterns in Object-Oriented Programming

- Design Patterns: Elements of Reusable Object-Oriented Software (1995)
  - Gang of Four (GOF): Gamma, Helm, Johnson, Vlissides
  - Catalogue of patterns
  - Creation, structural, behavioral
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

Here’s my algorithm. Where’s the concurrency?

MPEG Decoder

1. MPEG bit stream
2. VLD
   - macroblocks, motion vectors
3. split
   - frequency encoded macroblocks
   - differentially coded motion vectors
4. ZigZag
5. IQuantization
6. IDCT
7. Saturation
   - spatially encoded macroblocks
8. Motion Vector Decode
9. Repeat
10. join
    - motion vectors
11. Motion Compensation
    - recovered picture
12. Picture Reorder
13. Color Conversion
14. Display
Here’s my algorithm. Where’s the concurrency?

- **Task decomposition**
  - Independent coarse-grained computation
  - Inherent to algorithm

- **Sequence of statements (instructions) that operate together as a group**
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem
Here’s my algorithm. Where’s the concurrency?

- **Task decomposition**
  - Parallelism in the application

- **Data decomposition**
  - Same computation is applied to small data chunks derived from large data set
Here’s my algorithm. Where’s the concurrency?

- **Task decomposition**
  - Parallelism in the application

- **Data decomposition**
  - Same computation many data

- **Pipeline decomposition**
  - Data assembly lines
  - Producer-consumer chains
Guidelines for Task Decomposition

● Algorithms start with a good understanding of the problem being solved

● Programs often naturally decompose into tasks
  ■ Two common decompositions are
  – Function calls and
  – Distinct loop iterations

● Easier to start with many tasks and later fuse them, rather than too few tasks and later try to split them
Guidelines for Task Decomposition

- **Flexibility**
  - Program design should afford flexibility in the number and size of tasks generated
    - Tasks should not be tied to a specific architecture
    - Fixed tasks vs. Parameterized tasks

- **Efficiency**
  - Tasks should have enough work to amortize the cost of creating and managing them
  - Tasks should be sufficiently independent so that managing dependencies doesn’t become the bottleneck

- **Simplicity**
  - The code has to remain readable and easy to understand, and debug
Guidelines for Data Decomposition

- Data decomposition is often implied by task decomposition

- Programmers need to address task and data decomposition to create a parallel program
  - Which decomposition to start with?

- Data decomposition is a good starting point when
  - Main computation is organized around manipulation of a large data structure
  - Similar operations are applied to different parts of the data structure
Common Data Decompositions (1)

- Geometric data structures
  - Decomposition of arrays along rows, columns, blocks
  - Decomposition of meshes into domains

Figure from lecture by Prof. Mattan Erez, UT Austin.
Common Data Decompositions (2)

- Geometric data structures
  - Decomposition of arrays along rows, columns, blocks
  - Decomposition of meshes into domains

- Recursive data structures
  - Example: decomposition of trees into sub-trees
Guidelines for Data Decomposition

- **Flexibility**
  - Size and number of data chunks should support a wide range of executions

- **Efficiency**
  - Data chunks should generate comparable amounts of work (for load balancing)

- **Simplicity**
  - Complex data compositions can get difficult to manage and debug
Case for Pipeline Decomposition

- Data is flowing through a sequence of stages
  - Assembly line is a good analogy

- What’s a prime example of pipeline decomposition in computer architecture?
  - Instruction pipeline in modern CPUs

- What’s an example pipeline you may use in your UNIX shell?
  - Pipes in UNIX: `cat foobar.c | grep bar | wc`

- Other examples
  - Signal processing
  - Graphics
Re-engineering for Parallelism
Reengineering for Parallelism

- Parallel programs often start as sequential programs
  - Easier to write and debug
  - Legacy codes
  - Is this the right approach?

- How to reengineer a sequential program for parallelism
  - Survey the landscape
  - Pattern provides a list of questions to help assess existing code
  - Many are the same as in any reengineering project
  - Is program numerically well-behaved?
  - Is notion of correctness well defined for parallel execution?
    - Required precision of results
    - Input range
    - Performance expectations
  - Feasibility (back of envelope calculations)
Reengineering for Parallelism

- Define a testing protocol

- Identify program hot spots: where is most of the time spent?
  - Look at code
  - Use profiling tools

- Parallelization
  - Start with hot spots first
  - Make sequences of small changes, each followed by testing
  - Pattern provides guidance
Example: Molecular dynamics

- Simulate motion in large molecular system
  - Used for example to understand drug-protein interactions

- Forces
  - Bonded forces within a molecule
  - Long-range forces between atoms

- Naïve algorithm has $n^2$ interactions: not feasible

- Use cutoff method: only consider forces from neighbors that are “close enough”
Sequential Molecular Dynamics Simulator

// pseudo code
real[3,n] atoms
real[3,n] force
int [2,m] neighbors

function simulate(steps)
    for time = 1 to steps and for each atom
        Compute bonded forces
        Compute neighbors
        Compute long-range forces
        Update position
    end loop
end function
Finding Concurrency Design Space

- Decomposition Patterns
- Dependency Analysis Patterns
- Design Evaluation
Decomposition Patterns

- Main computation is a loop over atoms

- Suggests task decomposition
  - Task corresponds to a loop iteration
    - Update a single atom
  - Additional tasks
    - Calculate bonded forces
    - Calculate long range forces
  - Find neighbors
  - Update position

```
for time = 1 to steps and
  for each atom
    Compute bonded forces
    Compute neighbors
    Compute long-range forces
    Update position
end loop
```
Alternate Geometric Decomposition

- Molecular dynamics
  - Calculate forces
  - Update acceleration and velocities
  - Update positions

Figure from lecture by Prof. Mattan Erez, UT Austin.
No Free Lunch

- How the data is decomposed affects how the computation is carried out

- Still have to worry about boundary cases and overlap
  - Shared data dependences between tasks
  - Molecules that overlap two regions
Understand Control Dependences

- Bonded forces
- Neighbor list
- Long-range forces

Update position

next time step
Understand Data Dependences

Diagram:
- Bonded forces
- Neighbor list
- Long-range forces
- Update position
- Next time step

- atoms[3,n]
- forces[2,n]
- neighbors[2,m]

- Read
- Write
- Accumulate
Evaluate Design

- **What is the target architecture?**
  - Shared memory, distributed memory, message passing, ...

- Does data sharing have enough special properties (read only, accumulate, temporal constraints) that we can deal with dependences efficiently?

- If design seems OK, move to next design space
Programming For Parallelism Today

- Programmer controls every detail of parallelism

- Granularity decisions
  - If too small, lots of synchronization and thread creation
  - If too large, bad locality

- Load balancing decisions
  - Create balanced parallel sections (not data-parallel)
  - Profiling is a challenge

- Locality decisions
  - Code and data co-partitioning
  - Placement for sharing and optimized communication

- Synchronization decisions
  - Barriers, atomicity, critical sections, order, flushing, races, deadlocks

- Determinism nearly impossible
  - Debugging is heroic