Lecture Overview

- Thinking in Parallel
- Flynn’s Taxonomy
- Types of Parallelism
- Parallelism Basics
- Design Patterns for Parallel Programs
- Using GNU gprof
Parallelizing our lives

- Many of the tasks we perform in our everyday lives include significant parallelism.
- Can you name some of these?
- Discuss daily activities:
  - Identify at least 5-10 activities to get to work.
- Consider which of these activities could be carried out concurrently:
  - Identify pairs of parallelizable activities.
What is wrong with our world?

- Consider why many of these cannot presently be carried out in parallel
  - What would need to be changed in our physical world to allow us to complete many of these activities in parallel
  - How often is parallelism inhibited by our inability of carrying out things at same time?
- Estimate how much more quickly it would take to carry out activities if you could change these physical systems
Let’s Bake Some Cakes

- You are trying to bake 3 blueberry pound cakes
- Cake ingredients are as follows:
  - 1 cup butter, softened
  - 1 cup sugar
  - 4 large eggs
  - 1 teaspoon vanilla extract
  - 1/2 teaspoon salt
  - 1/4 teaspoon nutmeg
  - 1 1/2 cups flour 1 cup blueberries
The recipe for a single cake is as follows:

- **Step 1:** Preheat oven to 325°F (160°C). Grease and flour your cake pan.
- **Step 2:** In large bowl, beat together butter and sugar at medium speed until light and fluffy. Add eggs, vanilla, salt and nutmeg. Beat until thoroughly blended. Reduce mixer speed to low and add flour, 1/2 cup at a time, beating just until blended.
- **Step 3:** Gently fold in blueberries. Spread evenly in prepared baking pan. Bake for 35 minutes.
- **Step 4:** Cool for 20 minutes in pan. Remove from pan.
Your task is to cook 3 cakes as efficiently as possible. Assuming that you only have one oven large enough to hold one cake, one large bowl, one cake pan, and one mixer, come up with a schedule to make three cakes as quickly as possible. Identify the bottlenecks in completing this task.
Assume now that you have three bowls, 3 cake pans and 3 mixers. How much faster is the process now that you have additional resources?
• Assume now that you have two friends that will help you cook, and that you have three ovens that can accommodate all three cakes. How will this change the schedule you arrived at in Part a.) above?

• Compare the cake-making task to computing 3 iterations of a loop on a parallel computer. Identify data-level parallelism and task-level parallelism in the cake-making loop.
Lecture Overview

• Thinking in Parallel
• Flynn’s Taxonomy
• Types of Parallelism
• Parallelism Basics
• Design Patterns for Parallel Programs
• Using GNU gprof
Flynn’s Taxonomy of Arch.

- SISD - Single Instruction/Single Data
- SIMD - Single Instruction/Multiple Data
- MISD - Multiple Instruction/Single Data
- MIMD - Multiple Instruction/Multiple Data
The typical machine you’re used to (before multicores).

Single Instruction/Multiple Data

Processors that execute same instruction on multiple pieces of data.

Single Instruction/Multiple Data

• Each core executes same instruction simultaneously
• Vector-style of programming
• Natural for graphics and scientific computing
• Good choice for massively multicore
SISD versus SIMD

SIMD can be obtained through “intrinsics”, inline assembly, or automatically through compiler intervention.

Slide Source: ars technica, Peakstream article
Multiple Instruction/Single Data

Only Theoretical Machine. None ever implemented.

Many mainstream multicore processors fall into this category.
Multiple Instruction/Multiple Data

- Each core works independently, simultaneously executing different instructions on different data
- Unique upper levels of cache and may have lower level of shared cache or coherent cache
- Cores can have SIMD-extensions
- Programmed with a variety of models (OpenMP, MPI, pthreads, etc.)
Lecture Overview

- Thinking in Parallel
- Flynn’s Taxonomy
- Types of Parallelism
- Commercial Parallel Architectures
- Parallelism Basics
- Design Patterns for Parallel Programs
- Using GNU gprof
Types of Parallelism

Instructions:

- Pipelining
- Data-Level Parallelism (DLP)
- Thread-Level Parallelism (TLP)
- Instruction-Level Parallelism (ILP)

Slide Source: S. Amarasinghe, MIT 6189 IAP 2007
Pipelining

IF: Instruction fetch
EX: Execution
ID: Instruction decode
WB: Write back

Cycles

Instruction # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
---|---|---|---|---|---|---|---|---
Instruction i  | IF | ID | EX | WB |     |     |     |     
Instruction i+1 | IF | ID | EX | WB |     |     |     |     
Instruction i+2 | IF | ID | EX | WB |     |     |     |     
Instruction i+3 | IF | ID | EX | WB |     |     |     |     
Instruction i+4 | IF | ID | EX | WB |     |     |     |     

Corresponds to SI SD architecture.
Instruction-Level Parallelism

Dual instruction issue superscalar model. Again, corresponds to SISD architecture.

Slide Source: S. Amarasinghe, MIT 6189 IAP 2007
Data-Level Parallelism

Data Stream or Array Elements

What architecture model from Flynn’s Taxonomy does this correspond to?

Slide Source: Arch. of a Real-time Ray-Tracer, Intel
Data-Level Parallelism

Data Stream or Array Elements

[Diagram showing data stream or array elements]

Corresponds to SIMD architecture.

Slide Source: Arch. of a Real-time Ray-Tracer, Intel
One operation (e.g., +) produces multiple results. X, Y, and result are arrays.
Thread-Level Parallelism

Program partitioned into four threads. Four threads each executed on separate cores.

What architecture from Flynn’s Taxonomy does this correspond to?

Slide Source: SciDAC Review, Threadstorm pic.
Thread-Level Parallelism

Program partitioned into four threads. Four threads each executed on separate cores.

Corresponds to MIMD architecture.

Slide Source: SciDAC Review, Threadstorm pic.
Lecture Overview

- Thinking in Parallel
- Flynn’s Taxonomy
- Types of Parallelism
- Commercial Parallel Architectures
- Parallelism Basics
- Design Patterns for Parallel Programs
- Using GNU gprof
Generalized Multicore

Most nodes have dual-socket (two CPUs) with 12 cores per CPU.
Lecture Overview

- Thinking in Parallel
- Flynn’s Taxonomy
- Types of Parallelism
- Commercial Parallel Architectures
- Parallelism Basics
- Design Patterns for Parallel Programs
- Using GNU gprof
1. Study problem or code
2. Look for parallelism opportunities
3. Try to keep all cores busy doing useful work

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007
Decomposition

- Identify concurrency
  - Decide level to exploit
  - Requires understanding algorithm and data structures!
  - May require restructuring algorithm or writing an entirely new algorithm

- Break computation into tasks
  - Divided among processes
  - Tasks may become available dynamically
  - Number of tasks can vary with time

Want enough tasks to keep processors busy.

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007
Parallelization Common Steps

1. Study problem or code
2. Look for parallelism opportunities
3. Try to keep all cores busy doing useful work

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007
• Specify mechanism to divide work
• Balance of computation
• Reduce communication
• Structured approaches work well
  • Code inspection and understanding algorithm
  • Using design patterns (second half part of lecture)
Granularity

- Ratio of computation and communication
- **Fine-grain parallelism**
  - Tasks execute little comp. between comm.
  - Easy to load balance
  - If *too fine*, communication may take longer than computation
- **Coarse-grain parallelism**
  - Long computations between communication
  - More opportunity for performance increase
  - Harder to load balance

Most efficient granularity depends on algorithm/hardware.
Parallelization Common Steps

1. Study problem or code
2. Look for parallelism opportunities
3. Try to keep all cores busy doing useful work

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007
• Computation and communication concurrency
• Preserve locality of data
• Schedule task to satisfy dependences
Parallelization Common Steps

1. Study problem or code
2. Look for parallelism opportunities
3. Try to keep all cores busy doing useful work

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007
Lecture Overview

- Thinking in Parallel
- Flynn’s Taxonomy
- Types of Parallelism
- Commercial Parallel Architectures
- Parallelism Basics
- Design Patterns for Parallel Programs
- Using GNU gprof
Patterns for Parallelization

- Parallelization is a difficult problem
  - Hard to fully exploit parallel hardware

- Solution: *Design Patterns*

- Cookbook for parallel programmers
  - Can lead to high quality solutions

- Provides a common vocabulary
  - Each pattern has a name and associated vocabulary for discussing solutions

- Helps with software reusability and modularity
  - Allows developer to understand solution and its context
- Patterns for Parallel Programming.
  - Mattson et al. (2005)
- Four Design Spaces
  - Finding Concurrency
    - Expose concurrent task or data
  - Algorithm Structure
    - Map tasks to processes
  - Supporting Structures
    - Code and data structuring patterns
  - Implementation Mechanisms
    - Low-level mechanisms for writing programs
Finding Concurrency

- Decomposition
  - Task, Data, Pipeline
- Dependency Analysis
  - Control dependences
  - Data dependences
- Design Evaluation
  - Suitability for target platform
  - Design quality
Decomposition

- Data (domain) decomposition
  - Break data up into independent units
- Task (functional) decomposition
  - Break problem into parallel tasks
- Case for Pipeline decomposition
• Also known as Domain Decomposition
• Implied by Task Decomposition
  • *Which decomposition more natural to start with?*
    • 1) Decide how data is divided
    • 2) Decide how tasks should be performed
• Data decomposition is good starting point when
  • Main computation manipulating a large data structure
  • Similar operations applied to diff. parts of data structure
Common Data Decompositions

- Array-based computations
  - Decompose in a variety of ways including rows, columns, blocks
- Recursive-data structures
  - Example: Parallel updates of large tree, by decomposing into small trees
Find the largest element of an array
Data Decomposition

Find the largest element of an array

CPU 0  CPU 1  CPU 2  CPU 3

Slide Source: Intel Software College, Intel Corp.
Find the largest element of an array
Find the largest element of an array
Data Decomposition

Find the largest element of an array

CPU 0  CPU 1  CPU 2  CPU 3

Slide Source: Intel Software College, Intel Corp.
Find the largest element of an array
Data Decomposition

Find the largest element of an array

CPU 0 CPU 1 CPU 2 CPU 3

Slide Source: Intel Software College, Intel Corp.
Find the largest element of an array

CPU 0  CPU 1  CPU 2  CPU 3

Slide Source: Intel Software College, Intel Corp.
Data Decomposition

Find the largest element of an array

CPU 0  CPU 1  CPU 2  CPU 3

Slide Source: Intel Software College, Intel Corp.
Data Decomposition

Find the largest element of an array

CPU 0       CPU 1       CPU 2       CPU 3

Slide Source: Intel Software College, Intel Corp.
Find the largest element of an array

CPU 0  CPU 1  CPU 2  CPU 3

Slide Source: Intel Software College, Intel Corp.
Find the largest element of an array

CPU 0     CPU 1     CPU 2     CPU 3

Slide Source: Intel Software College, Intel Corp.
Data Decomposition Forces

- **Flexibility**
  - Parameterize size of data chunks to support a range of systems
    - Granularity knobs
- **Efficiency**
  - Data chunks should have comparable computation
    - Load balancing
- **Simplicity**
  - Complex data decomposition difficult to debug
Programs often naturally decompose into tasks
  - Functions
  - Divide tasks among cores
    - Easier to start with too many task and fuse some later
  - Decide data accessed (read/written) by each core
Task Decomposition

Slide Source: Intel Software College, Intel Corp.
Task Decomposition

Slide Source: Intel Software College, Intel Corp.
Task Decomposition

Slide Source: Intel Software College, Intel Corp.
Task Decomposition

Slide Source: Intel Software College, Intel Corp.
Task Decomposition

Slide Source: Intel Software College, Intel Corp.
Task Decomposition

Slide Source: Intel Software College, Intel Corp.
Task Decomposition Forces

- Flexibility in number and size of tasks
  - Task should not be tied to a specific architecture
  - Parameterize number of task
    - Flexible to any architecture topology
- Efficiency
  - Task have enough computation to amortize creation costs
  - Sufficiently independent so dependencies are manageable
- Simplicity
  - Easy to understand and debug

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Special kind of task decomposition
  - Data flows through a sequence of tasks
- “Assembly line” parallelism
- Example: 3D rendering in computer graphics

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Processing one data set (Step 1)
Pipeline Decomposition

- Processing one data set (Step 2)

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Processing one data set (Step 3)
Pipeline Decomposition

- Processing one data set (Step 4)
  - Pipeline processes 1 data set in 4 steps

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

• Processing five data set (Step 1)

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Processing five data set (Step 2)

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Processing five data set (Step 3)

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Processing five data set (Step 4)

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Processing five data set (Step 5)

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Processing five data set (Step 6)

Diagram showing processing stages for four CPUs:
- CPU 0:
  - Data set 0
  - Data set 1
  - Data set 2
  - Data set 3
  - Data set 4
- CPU 1:
  - Data set 0
  - Data set 1
  - Data set 2
  - Data set 3
  - Data set 4
- CPU 2:
  - Data set 0
  - Data set 1
  - Data set 2
  - Data set 3
  - Data set 4
- CPU 3:
  - Data set 0
  - Data set 1
  - Data set 2
  - Data set 3
  - Data set 4

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

- Processing five data set (Step 7)

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition

• Processing five data set (Step 8)

Slide Source: Intel Software College, Intel Corp.
Pipeline Decomposition Forces

- **Flexibility**
  - Deeper pipelines are better
    - Will scale better and can later merge pipeline stages

- **Efficiency**
  - Stages of pipeline should not cause bottleneck
    - Slowest stage is bottleneck

- **Simplicity**
  - More pipeline stages break down problem into more manageable chunks of code
Dependency Analysis

• Control and Data Dependences

• Dependence Graph
  • Graph = (nodes, edges)
  • Node for each
    • Variable assignment
    • Constant
    • Operator or Function call
  • Edge indicates use of variables and constants
    • Data flow
    • Control flow
for (i = 0; i < 3; i++)
a[i] = b[i] / 2.0;
for (i = 0; i < 3; i++)
a[i] = b[i] / 2.0;

Domain decomposition possible

Slide Source: Intel Software College, Intel Corp.
for (i = 1; i < 4; i++)
    a[i] = a[i-1] * b[i];
for (i = 1; i < 4; i++)
a[i] = a[i-1] * b[i];

No domain decomposition
a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;
a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;

Task decomposition with 3 CPUs.
Lecture Overview

- Thinking in Parallel
- Flynn’s Taxonomy
- Types of Parallelism
- Parallelism Basics
- Design Patterns for Parallel Programs
- Using GNU gprof
GNU gprof time profiler

- Detail time statistics for each subroutine.
- Create relative graph for all subroutines.
- Analysis the program bottleneck.
- Increase about 30% extra time cost.

Adapted from:
http://www.math.ntu.edu.tw/~wwang/cola_lab/knowledge/download/gprof/GNU_gprof_Profile.ppt
Recompile the original source code

```bash
gcc -pg SourceCode -o ExecutableFile
$ gcc -pg test2.c -o test2
```

- `-pg`: This option affects both compiling and linking.

Add additional commands into source code when compiling code in order to trace all subroutines.

Add essential initial settings and statistical processes when linking the objects.
- Convert produced profile data into text file

```
gprof ListOfOptions ExecuteFile StatFiles > OutputFile
$ gprof -b test2 gmon.out > output.txt
```

- `ListOfOptions` can be omitted.
- `ExecuteFile` can be omitted when the file name is `.a.out`.
- `StatFiles` can be omitted when the file name is `.gmon.out`. 
GNU gprof time profiler

- List of Options

- `b`: brief description, i.e., omit the table or data illustration

- `e (E) func-name`: exclude the subroutine `func-name` from the table (and exclude its elapsed time).

- `f (F) func-name`: only display the subroutine `func-name` on the table (and its elapsed time).
List of Options

- `s`: combine more than one StatFile into single one with default file name `gmon.sum`.

- `z`: only display all subroutines table which are unused on the program.
• Example Program

subroutine relative graph
Example Program

$ gcc -pg test.c -o test test
$ ./test
gprof -b test gmon.out > output
$ more output
Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>time</th>
<th>self seconds</th>
<th>s/call</th>
<th>total seconds</th>
<th>s/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>71.90</td>
<td>30.17</td>
<td>30.17</td>
<td>30.17</td>
<td>1</td>
<td>30.17</td>
<td>C3</td>
</tr>
<tr>
<td>19.42</td>
<td>38.32</td>
<td>8.15</td>
<td>4.07</td>
<td>2</td>
<td>4.07</td>
<td>B2</td>
</tr>
<tr>
<td>7.99</td>
<td>41.67</td>
<td>3.35</td>
<td>3.35</td>
<td>1</td>
<td>3.35</td>
<td>C2</td>
</tr>
<tr>
<td>0.00</td>
<td>41.67</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>A</td>
</tr>
<tr>
<td>0.00</td>
<td>41.67</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>B1</td>
</tr>
<tr>
<td>0.00</td>
<td>41.67</td>
<td>0.00</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>C1</td>
</tr>
<tr>
<td>0.00</td>
<td>41.67</td>
<td>0.00</td>
<td>4.07</td>
<td>1</td>
<td>4.07</td>
<td>D</td>
</tr>
</tbody>
</table>
Example Program

% time: the percent of self seconds from total program elapsed time.

cumulative seconds: the seconds cumulate from self seconds.

self seconds: total elapsed time called by its parents, not including its children’s elapsed time. equal to (self s/call)*(calls)
Example Program

calls: total number for each subroutine called by its parents.

self s/call: elapsed time for each time called by its parents, not including its children’s elapsed time.

total s/call: total elapsed time called by its parents, including its children’s elapsed time.

name: subroutine name.
Organize by Data

- Operations on core data structure
- Geometric Decomposition
- Recursive Data
Geometric Decomposition

- Arrays and other linear structures
  - Divide into contiguous substructures
- Example: Matrix multiply
  - Data-centric algorithm and linear data structure (array) implies geometric decomposition
Recursive Data

- Lists, trees, and graphs
  - Structures where you would use divide-and-conquer
- May seem that can only move sequentially through data structure
  - But, there are ways to expose concurrency
Organize by Flow of Data

- Regular
  - Pipeline
- Irregular
  - Event-Based Coordination
Organize by Flow of Data

- Computation can be viewed as a flow of data going through a sequence of stages
- Pipeline: one-way predictable communication
- Event-based Coordination: unrestricted unpredictable communication
Pipeline performance

- Concurrency limited by pipeline depth
  - Balance computation and communication (architecture dependent)
- Stages should be equally computationally intensive
  - Slowest stage creates bottleneck
  - Combine lightly loaded stages or decompose heavily-loaded stages
- Time to fill and drain pipe should be small
Supporting Structures

- Single Program Multiple Data (SPMD)
- Loop Parallelism
- Master/Worker
- Fork/Join
SPMD Pattern

- Create single program that runs on each processor
  - Initialize
  - Obtain a unique identifier
  - Run the same program each processor
    - Identifier and input data can differentiate behavior
  - Distribute data (if any)
  - Finalize

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007
SPMD Challenges

- Split data correctly
- Correctly combine results
- Achieve even work distribution
- If programs require dynamic load balancing, another pattern may be more suitable (Job Queue)
Loop Parallelism Pattern

• Many programs expressed as iterative constructs
• Programming models like OpenMP provide pragmas to automatically assign loop iterations to processors

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

Independent Tasks

master

A B C D E

worker

A B C E D

Slide Source: Dr. Rabbah, IBM, MIT Course 6.189 IAP 2007
• Relevant where tasks have no dependencies
  • Embarrassingly parallel
• Problem is determining when entire problem complete
Fork/Join Pattern

- Parent creates new tasks (fork), then waits until they complete (join)
- Tasks created dynamically
  - Tasks can create more tasks
- Tasks managed according to relationships