“Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.”

Isaac Asimov
Before We Get Started...

- **Last Time**
  - OpenMP wrap up
    - Variable scoping
    - Synchronization issues

- **Today**
  - Parallel programming patterns
  - One slide summary of ME964

- **Other issues:**
  - No class on April 12
  - Assignment 7 due tonight
  - Assignment 8 (last ME964 assignment) posted on the class website, due next Th
  - Midterm exam on April 19
    - Review session the evening before
  - From now on only guest lectures and such, time to concentrate on your projects
Recommended Reading


- Used for this presentation
- A good overview of challenges, best practices, and common techniques in all aspects of parallel programming
- Book is on reserve at Wendt Library
Objective

- Get exposed to techniques & best practices that have emerged as useful in the parallel programming practice
- They are expected to facilitate and/or help you with
  - Thinking/Visualizing your problem as being solved in parallel
  - Addressing functionality and performance issues in your parallel program design
  - Discussing your design decisions with others
  - Selecting an appropriate platform that helps you express & implement the parallel design for the solution you identified
Parallel Computing: When and Why.

- Parallel computing, prerequisites
  - The problem can be decomposed into sub-problems that can be independently solved at the same time
  - The part of the problem that is concurrent is large enough to justify an approach that exploits this concurrency
    - Recall Amdhal’s law

- Parallel computing, goals
  - Solve problems in less time
  - Solve bigger problems
Performance can be drastically reduced by many factors

- Overhead of parallel processing
- Load imbalance among processor elements
- Inefficient data sharing patterns
- Saturation of critical resources such as memory bandwidth
Implementing a Parallel Solution to Your Problem: Key Steps

1) Find the concurrency in the problem

2) Structure the algorithm so that concurrency can be exploited

3) Implement the algorithm in a suitable programming environment

4) Tune the performance of the code on the target parallel system

NOTE: The reality is that these have not been separated into levels of abstractions that can be dealt with independently.
What’s Next?

Focus on this for a while

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

From "Patterns for Parallel Programming"
Finding and Nurturing Concurrency

- Dependence kills concurrency. Dependencies need to be identified and managed
  - Dependencies prevent parallelism, at least the “embarrassingly parallel” flavor
  - Often times, dependencies end up requiring synchronization barriers (not good…)

- Concurrency has some caveats
  - In sequential execution: One step feeds result to the next steps ⇒ a unique way moving from A to Z
  - In parallel execution: numeric accuracy may be affected by ordering steps that parallel with each other ⇒ platform dependent, OS dependent possibly even dependent on the state of the system

- Finding and exploiting concurrency often requires looking at the problem from a non-obvious angle
Finding Concurrency in Problems

- **Goal:** Identify a decomposition of the problem into sub-problems that can be solved simultaneously

- **In order to meet this goal:**
  - Perform a *task decomposition* to identify tasks that can execute concurrently
  - Carry out *data decomposition* to identify data local to each task
  - Identify a way of *grouping* tasks and *ordering* the groups to satisfy temporal constraints
  - Carry out an analysis of the data *sharing patterns* among the concurrent tasks to avoid any race condition issues and optimize memory access
  - Perform a *design evaluation* that assesses the quality of the choices made in all the steps
Finding Concurrency – The Process

This is typically an iterative process, like an optimization process that has to negotiate several constraints.
Find Concurrency 1: Decomp. Stage: Task Decomposition

- Many large problems have natural independent tasks
  - The number of tasks used should be adjustable to the execution resources available.
  - Each task must include sufficient work in order to compensate for the overhead of managing their parallel execution.
  - Tasks should maximize reuse of sequential program code to minimize design/implementation effort.

“In an ideal world, the compiler would find tasks for the programmer. Unfortunately, this almost never happens.”
- Mattson, Sanders, Massingill
Example: Task Decomposition
Square Matrix Multiplication

- **P = M * N of WIDTH ● WIDTH**
  - One natural task (sub-problem) produces one element of P
  - All tasks can execute in parallel
Find Concurrency 2: Decomp. Stage: Data Decomposition

- The most compute intensive parts of many large problems manipulate a large data structure
  - Similar operations are being applied to different parts of the data structure, in a mostly independent manner.
  - This is what CUDA is optimized for.

- The data decomposition should lead to
  - Efficient data usage by each Unit of Execution (UE) within the partition
  - Few dependencies between the UEs that work on different partitions
  - Adjustable partitions that can be varied according to the hardware characteristics
sometimes several tasks in problem can be grouped to improve efficiency

- reduced synchronization overhead – because when task grouping there is supposedly no need for synchronization
- all tasks in the group efficiently share data loaded into a common on-chip, shared storage (shared memory)
Task Grouping Example - Square Matrix Multiplication

- Tasks calculating a $P_{\text{sub}}$ block
  - Extensive input data sharing, reduced memory bandwidth using Shared Memory
  - All synched in execution
Find Concurrency 4: Dependence Analysis - Task Ordering

- Identify the data required by a group of tasks before they can execute
  - Find the task group that creates it (look upwind)
  - Determine a temporal order that satisfy all data constraints
Task Ordering Example: Finite Element Analysis

1. **Node List**
2. **External Forces**
3. **Element Internal Force**
4. **Acceleration of each deformable beam**
5. **Update position and velocity of each beam**
6. **Next Time Step**
Data sharing can be a double-edged sword

- An algorithm that calls for excessive data sharing can drastically reduce advantage of parallel execution

- Localized sharing can improve memory bandwidth efficiency

- Use the execution of task groups to interleave with (mask) global memory data accesses

- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires a higher level of synchronization
Data Sharing Example
Matrix Multiplication on the GPU

- Each task group will finish usage of each sub-block of N and M before moving on
  - N and M sub-blocks loaded into Shared Memory for use by all threads of a P sub-block
  - Amount of on-chip Shared Memory strictly limits the number of threads working on a P sub-block

- Read-only shared data can be efficiently accessed as Constant or Texture data (on the GPU)
  - Frees up the shared memory for other uses
Find Concurrency 6: Design Evaluation

- Key questions to ask
  - How many Units of Execution (UE) can be used?
  - How are the data structures shared?
  - Is there a lot of data dependency that leads to excessive synchronization needs?
  - Is there enough work in each UE between synchronizations to make parallel execution worthwhile?
Implementing a Parallel Solution to Your Problem: Key Steps

1) Find the concurrency in the problem

2) Structure the algorithm so that concurrency can be exploited

3) Implement the algorithm in a suitable programming environment

4) Execute and tune the performance of the code on a parallel system
What’s Next?

Focus on this for a while

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

From “Patterns for Parallel Programming”
Algorithm: Definition

- A step by step procedure that is guaranteed to terminate, such that each step is precisely stated and can be carried out by a computer
  - Definiteness – the notion that each step is precisely stated
  - Effective computability – each step can be carried out by a computer
  - Finiteness – the procedure terminates
- Multiple algorithms can be used to solve the same problem
  - Some require fewer steps and some exhibit more parallelism
Algorithm Structure - Strategies

Organize by Tasks
- Task Parallelism
- Divide and Conquer

Organize by Data
- Geometric Decomposition
- Recursive Data Decomposition

Organize by Flow
- Pipeline
- Event Condition
Choosing Algorithm Structure

Algorithm Design

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

Organize by Data
- Linear
  - Geometric Decomposition
- Recursive
  - Recursive Data Decmp.

Organize by Data Flow
- Regular
  - Pipeline
- Irregular
  - Event Driven
Alg. Struct. 1: Organize by Structure
Linear Parallel Tasks

- Common in the context of distributed memory models

- These are the algorithms where the work needed can be represented as a collection of decoupled or loosely coupled tasks that can be executed in parallel
  - The tasks don’t even have to be identical
  - Load balancing between UE can be an issue (dynamic load balancing?)
  - If there is no data dependency involved there is no need for synchronization: the so called “embarrassingly parallel” scenario
Alg. Struct. 1: Organize by Structure
Linear Parallel Tasks [Cntd.]

- Examples
  - Imagine a car that needs to be painted:
    - One robot paints the front left door, another one the rear left door, another one the hood, etc.
    - The car is parceled up with a collection of UEs taking care of subtasks
  - Other: ray tracing, a good portion of the N-body problem, Monte Carlo type simulation
Alg. Struct. 2: Organize by Structure
Recursive Parallel Tasks (Divide & Conquer)

- Valid when you can break a problem into a set of decoupled or loosely coupled smaller problems, which in turn can be broken etc…
  - This pattern is applicable when you can solve concurrently and with little synchronization the small problems (the leafs)

- In some case you need synchronization when dealing with this balanced tree type algorithm
  - Often required by the merging step (assembling the result from the “subresults”)

- Examples: FFT, Linear Algebra problems (see FLAME project), the vector reduce operation
Computational ordering can have major effects on memory bank conflicts and control flow divergence.
Alg. Struct. 3: Organize by Data
~ Linear (Geometric Decomposition) ~

- This is the case where the UEs gather and work around a big chunk of data with little or no synchronization.

- This is exactly the algorithmic approach enabled best by the GPU & CUDA.

- Examples: Matrix multiplication, matrix convolution, image processing.
This scenario comes into play when the data that you operate on is structured in a recursive fashion
- Balanced Trees
- Graphs
- Lists

Sometimes you don’t even have to have the data represented as such
- See example with prefix scan
- You choose to look at data as having a balanced tree structure

Problems that seem inherently sequential can be approached in this framework
- This is typically associated with an net increase in the amount of work you have to do
- Work goes up from $O(n)$ to $O(n \log(n))$ (see for instance Hillis and Steele algorithm)
- The key question is whether parallelism gained brings you ahead of the sequential alternative
Example: The Prefix Scan
~ Reduction Step~

\[
\begin{align*}
&\text{for } k=0 \text{ to } M-1 \\
&\quad \text{offset} = 2^k \\
&\quad \text{for } j=1 \text{ to } 2^{M-k-1} \text{ in parallel do} \\
&\quad \quad x[j \cdot 2^{k+1}-1] = x[j \cdot 2^{k+1}-1] + x[j \cdot 2^{k+1}-2^k-1] \\
&\quad \text{endfor} \\
&\text{endfor}
\end{align*}
\]
Example: The array reduction (the bad choice)

Array elements

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1

<table>
<thead>
<tr>
<th>0+1</th>
<th>2+3</th>
<th>4+5</th>
<th>6+7</th>
<th>8+9</th>
<th>10+11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2

<table>
<thead>
<tr>
<th>0...3</th>
<th>4..7</th>
<th>8..11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3

<table>
<thead>
<tr>
<th>0..7</th>
<th>8..15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alg. Struct. 5: Organize by Data Flow
~ Regular: Pipeline ~

- Tasks are parallel, but there is a high degree of synchronization (coordination) between the UEIs
  - The input of one UE is the output of an upwind UE (the “pipeline”)
  - The time elapsed between two pipeline ticks dictated by the slowest stage of the pipeline (the bottleneck)

- Commonly employed by sequential chips

- For complex problems having a deep pipeline is attractive
  - At each pipeline tick you output one set of results

- Examples:
  - CPU: instruction fetch, decode, data fetch, instruction execution, data write are pipelined.
  - The Cray vector architecture drawing heavily on this algorithmic structure when performing linear algebra operations
Alg. Struct. 6: Organize by Data Flow
~ Irregular: Event Driven Scenarios ~

- The furthest away from what the GPU can support today
- Well supported by MIMD architectures
- You coordinate UEs through asynchronous events

- Critical aspects:
  - Load balancing – should be dynamic
  - Communication overhead, particularly in real-time applications

- Suitable for action-reaction type simulations

- Examples: computer games, traffic control algorithms, server operation (amazon, google)
Implementing a Parallel Solution to Your Problem:
Key Steps

1) Find the concurrency in the problem

2) Structure the algorithm so that concurrency can be exploited

3) **Implement the algorithm in a suitable programming environment**

4) Execute and tune the performance of the code on a parallel system
What’s Comes Next?

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

Focus on this for a while
Above are the models for which parallel software/hardware combos provide good support nowadays. If you don’t fall in one of the above there’ll be no sailing, you’ll have to row.
Data Models

- Shared Data
  - All threads share a major data structure
  - This is what CUDA and GPU computing support the best

- Shared Queue
  - All threads see a “thread safe” queue
  - Very relevant in conjunction with the Master/Worker scenarios
  - OpenMP is very helpful here if your problem fits on one machine
    - If not, MPI can help

- Distributed Array
  - Decomposed and distributed among threads
  - Limited support in CUDA Shared Memory but the direction where libraries are going (thrust, for instance)
  - Good library support under MPI (this is how things get done in PETSc)
  - OpenMP: doesn’t apply
**Program Models**

- **Master/Worker**
  - A Master thread sets up a pool of worker threads and a bag of tasks
  - Workers execute concurrently, removing tasks until done
  - Common in OpenMP

- **Loop Parallelism**
  - Loop iterations execute in parallel
  - FORTRAN do-all (truly parallel), do-across (with dependence)
  - Very common in OpenMP

- **Fork/Join**
  - Most general way of creation of threads (the POSIX standard)
  - Can be regarded as a very low level approach in which you use the OS to manage parallelism
Program Models

- SPMD (Single Program, Multiple Data)
  - All PE’s (Processor Elements) execute the same program in parallel
  - Each PE has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PE can follow different paths through the same code

SPMD is by far the most commonly used pattern for structuring parallel programs.
More on SPMD

- Dominant coding style of scalable parallel computing
  - MPI code is mostly developed in SPMD style
  - Almost exclusively used as the pattern in GPU computing
  - Much OpenMP code is also in SPMD
  - Particularly suitable for algorithms based on data parallelism, geometric decomposition, divide and conquer.

- Main advantage
  - Tasks and their interactions visible in one piece of source code, no need to correlate multiple sources
Typical SPMD Program Phases

- Initialize
  - Establish localized data structure and communication channels

- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically in the range from 0 to N-1, where N is the number of threads.
    - OpenMP, MPI, and CUDA have built-in support for this

- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads

- Run the core computation
  - More details in next slide…

- Finalize
  - Reconcile global data structure, prepare for the next major iteration
Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
  - CUDA: `Indx.x`, `Indx.y`, `Indx.z` (also block ids)
  - MPI: rank of a process
  - OpenMP: `get_thread_num()` – gets id associated with a specific thread in a parallel region

- Use thread ID in loop index calculations to split loop iterations among threads

- Use conditions based on thread ID to branch to their specific actions
### Algorithm Structures [in columns] vs. Program Models [in rows]

<table>
<thead>
<tr>
<th></th>
<th>Task Parallel.</th>
<th>Divide/Conquer</th>
<th>Geometric Decomp.</th>
<th>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>

Four smilies is the best (Source: Mattson, et al.)
## Parallel Programming Support vs. Program Models

<table>
<thead>
<tr>
<th>Program Models</th>
<th>OpenMP</th>
<th>MPI</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
</tr>
<tr>
<td>Loop Parallel</td>
<td>😊😊😊</td>
<td>😊</td>
<td></td>
</tr>
<tr>
<td>Master/Slave</td>
<td>😊😊</td>
<td>😊😊😊</td>
<td></td>
</tr>
<tr>
<td>Fork/Join</td>
<td>😊😊😊</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*[Note: The symbols represent the level of support: 😊 = high, 😊😊 = moderate, 😊😊😊 = low]
Sequential computing hit three walls: power, memory, and ILP walls

Moore’s law scales at least for one more decade
  - Parallel computing poised to pick up where sequential computing left

Moore’s law brought us powerful CPUs and GPUs
  - Use OpenMP and/or CUDA (or OpenCL) to leverage this hardware

Large problems do not always fit inside one workstation
  - Not enough memory, or if enough memory not enough crunching number power

Large problem handled well by clusters or massively parallel computers
  - MPI helps you here

When possible, use parallel libraries (thrust, PETSc, TBB, MKL, etc.)
  - However, writing your own code for your own small problem sometimes pays off nicely