Term Exam Format

- 3 sections
  - Parallel Analysis
  - Parallel Architecture
  - Parallel Algorithms and Programming

- Each section consists of 2 parts:
  - Short answers
  - Problems

- Time: 90 minutes
  - Approximately 30 minutes for each section
Term Exam Content

- Covered a lot of area in lecture and in labs
- Can not ask questions on everything
- Attempt to test your general acquired knowledge
  - Some parts will be more detailed
- Exam should be a learning experience!
  - But you will learn most in preparing for the exam
  - Best strategy is to read through the lecture slides
- Exam is open book and open notes
- Computers only to look at lecture slides or notes
  - No searching allowed
What will not be on the exam …

- No question on iso efficiency
- No writing code in Cilk Plus or TBB
Performance and Scalability

- Evaluation
  - **Sequential** runtime ($T_{seq}$) is a function of
    - problem size and architecture
  - **Parallel** runtime ($T_{par}$) is a function of
    - problem size and parallel architecture
    - # processors used in the execution
  - Parallel performance affected by
    - algorithm + architecture

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
Performance Metrics and Formulas

- $T_1$ is the execution time on a single processor
- $T_p$ is the execution time on a $p$ processor system
- $S(p)$ ($S_p$) is the speedup

\[
S(p) = \frac{T_1}{T_p}
\]

- $E(p)$ ($E_p$) is the efficiency

\[
Efficiency = \frac{S_p}{p}
\]

- $Cost(p)$ ($C_p$) is the cost

\[
Cost = p \times T_p
\]

- Parallel algorithm is cost-optimal
  - Parallel time = sequential time ($C_p = T_1$, $E_p = 100\%$)
Amdahl’s Law (Fixed Size Speedup)

- Let $f$ be the fraction of a program that is sequential
  - $1-f$ is the fraction that can be parallelized
- Let $T_1$ be the execution time on 1 processor
- Let $T_p$ be the execution time on $p$ processors
- $S_p$ is the speedup
  \[ S_p = \frac{T_1}{T_p} = \frac{T_1}{fT_1 + (1-f)T_1/p}) = \frac{1}{f + (1-f)/p}) \]
- As $p \rightarrow \infty$
  \[ S_p = \frac{1}{f} \]
Amdahl’s Law and Scalability

- **Scalability**
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

- **When does Amdahl’s Law apply?**
  - When the problem size is fixed
  - *Strong scaling* \( (p \to \infty, S_p = S_\infty \to 1 / f) \)
  - Speedup bound is determined by the degree of sequential execution time in the computation, not # processors!!!
  - Uhh, this is not good … Why?
  - Perfect efficiency is hard to achieve

- See original paper by Amdahl on webpage
Gustafson-Barsis’ Law (Scaled Speedup)

- Often interested in larger problems when scaling
  - How big of a problem can be run (HPC Linpack)
  - Constrain problem size by parallel time

- Assume parallel time is kept constant
  - \( T_p = C = (f + (1-f)) \times C \)
  - \( f_{seq} \) is the fraction of \( T_p \) spent in sequential execution
  - \( f_{par} \) is the fraction of \( T_p \) spent in parallel execution

- What is the execution time on one processor?
  - Let \( C=1 \), then
    \[
    T_s = f_{seq} + p(1-f_{seq}) = 1 + (p-1)f_{par}
    \]

- What is the speedup in this case?
  - \( S_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par} \)

\( T_s \) is determined by the problem size with \( P \) processors
Gustafson-Barsis’ Law and Scalability

- **Scalability**
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

- **When does Gustafson’s Law apply?**
  - When the problem size can increase as the number of processors increases
  - *Weak scaling* \( S_p = 1 + (p-1)f_{par} \)
  - Speedup function includes the number of processors!!!
  - Can maintain or increase parallel efficiency as the problem scales

- See original paper by Gustafson on webpage
Amdahl versus Gustafson-Baris

Amdahl

P=1

serial work

parallelizable work

Time

P=2

P=4

P=8
Amdahl versus Gustafson-Baris

Gustafson-Baris

- Serial work
- Parallelizable work

<table>
<thead>
<tr>
<th>P=1</th>
<th>P=2</th>
<th>P=4</th>
<th>P=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DAG Model of Computation

- Think of a program as a directed acyclic graph (DAG) of tasks
  - A task can not execute until all the inputs to the tasks are available
  - These come from outputs of earlier executing tasks
  - DAG shows explicitly the task dependencies
- Think of the hardware as consisting of workers (processors)
- Consider a greedy scheduler of the DAG tasks to workers
  - No worker is idle while there are tasks still to execute
Work-Span Model

- $T_P = \text{time to run with } P \text{ workers}$
- $T_1 = \text{work}$
  - Time for serial execution
    - execution of all tasks by 1 worker
  - Sum of all work
- $T_\infty = \text{span}$
  - Time along the critical path
- Critical path
  - Sequence of task execution (path) through DAG that takes the longest time to execute
  - Assumes an infinite # workers available
Lower/Upper Bound on Greedy Scheduling

- Suppose we only have $P$ workers
- We can write a work-span formula to derive a lower bound on $T_P$
  - $\max(T_1 / P, T_\infty) \leq T_P$
- $T_\infty$ is the best possible execution time
- Brent’s Lemma derives an upper bound
  - Capture the additional cost executing the other tasks not on the critical path
  - Assume can do so without overhead
  - $T_P \leq (T_1 - T_\infty) / P + T_\infty$
Estimating Running Time

- Scalability requires that $T_\infty$ be dominated by $T_1$
  $$T_P \approx T_1 / P + T_\infty \text{ if } T_\infty << T_1$$
- The span impacts scalability, even for finite $P$
- Sufficient parallelism implies linear speedup

\[
T_P \approx T_1 / P \quad \text{if} \quad T_1 / T_\infty \gg P
\]

Linear speedup

Parallel slack
Parallelism, Correctness, and Dependence

- Parallel execution, from any point of view, will be constrained by the sequence of operations needed to be performed for a correct result.
- Parallel execution must address control, data, and system dependences.
- A *dependency* arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed.
- We extend this notion of dependency to resources since some operations depend on certain resources.
Executing Two Statements in Parallel

- Want to execute two statements in parallel
- On one processor:
  - Statement 1;
  - Statement 2;
- On two processors:
  - Processor 1:
  - Statement 1;
  - Processor 2:
  - Statement 2;

- Fundamental (concurrent) execution assumption
  - Processors execute independent of each other
  - No assumptions made about speed of processor execution
Sequential Consistency in Parallel Execution

- **Case 1:**
  - Processor 1: statement 1;
  - Processor 2: time
  - statement 2;

- **Case 2:**
  - Processor 1: time
  - Processor 2: statement 2;
  - statement 1;

- **Sequential consistency**
  - Statements execution does not interfere with each other
  - Computation results are the same (independent of order)
Independent versus Dependent

- In other words the execution of
  
  `statement1;`  
  `statement2;`  
  
  must be equivalent to  
  
  `statement2;`  
  `statement1;`  

- Their order of execution must not matter!
- If true, the statements are *independent* of each other
- Two statements are *dependent* when the order of their execution affects the computation outcome
Examples

- **Example 1**
  S1: a=1;
  S2: b=1;
  - Statements are independent

- **Example 2**
  S1: a=1;
  S2: b=a;
  - Dependent (*true (flow) dependence*)
    - Second is dependent on first
    - Can you remove dependency?

- **Example 3**
  S1: a=f(x);
  S2: a=b;
  - Dependent (*output dependence*)
    - Second is dependent on first
    - Can you remove dependency? How?

- **Example 4**
  S1: a=b;
  S2: b=1;
  - Dependent (*anti-dependence*)
    - First is dependent on second
    - Can you remove dependency? How?
When can two statements execute in parallel?

- Statements S1 and S2 can execute in parallel if and only if there are *no dependences* between S1 and S2
  - True dependences
  - Anti-dependences
  - Output dependences
- Some dependences can be removed by modifying the program
  - Rearranging statements
  - Eliminating statements
**Parallel Patterns**

- Understand the different parallel patterns discussed in lecture and in labs

- Parallel control patterns: fork-join, map, stencil reduction, scan, recurrence

- Parallel data management patterns: pack, pipeline, geometric decomposition gather, scatter

- Understand the associated parallel execution issues
Parallelizing Algorithms

- Common strategy:
  - Divide up the computational domain into sections
  - Work on the sections individually
  - Combine the results

- Methods
  - Divide-and-conquer
  - Fork-join (discussed in Chapter 8)
  - Geometric decomposition
  - Partitions
  - Segments
Partitioning

- Data is divided into
  - non-overlapping regions (avoid write conflicts, race conditions)
  - equal-sized regions (improve load balancing)
A stencil pattern is a map where each output depends on a “neighborhood” of inputs.

These inputs are a set of fixed offsets relative to the output position.

A stencil output is a function of a “neighborhood” of elements in an input collection.

- Applies the stencil to select the inputs.

Data access patterns of stencils are regular.

- Stencil is the “shape” of “neighborhood”
- Stencil remains the same.
Iterative Codes

- Iterative codes are ones that update their data in steps
  - At each step, a new value of an element is computed using a formula based on other elements
  - Once all elements are updated, the computation proceeds to the next step or completes

- Iterative codes are most commonly found in computer simulations of physical systems for scientific and engineering applications
  - Computational fluid dynamics
  - Electromagnetics modeling

- They are often applied to solve partial differential equations
  - Jacobi iteration
  - Gauss-Seidel iteration
  - Successive over relaxation
Iterative Codes and Stencils

- Stencils essentially define which elements are used in the update formula.
- Because the data is organized in a regular manner, stencils can be applied across the data uniformly.
Conway’s Game of Life

- The Game of Life computation can easily fit into the stencil pattern!
- Each larger, black box is owned by a thread
- What will happen at the boundaries?
Conway’s Game of Life

- We need some way to preserve information from the previous iteration without overwriting it.
- **Ghost Cells** are one solution to the boundary and update issues of a stencil computation.
- Each thread keeps a copy of neighbors’ data to use in its local computations.
- These ghost cells must be updated after each iteration of the stencil.
Stencil and Communication Optimizations

- Generally better to replicate ghost cells in each local memory and swap after each iteration than to share memory
  - Fine-grained sharing can increase communication cost
- **Halo**: set of all ghost cells
  - Halo must contain all neighbors needed for one iteration
- Larger halo (**deep halo**)
  - Trade off
    - less communications and more independence, but…
    - more redundant computation and more memory used
- **Latency Hiding**:
  - Compute interior of stencil waiting for ghost cell updates
Fork-Join Concept

- Fork-Join is a fundamental way (primitive) of expressing concurrency within a computation

- **Fork** is called by a (logical) thread (*parent*) to create a new (logical) thread (*child*) of concurrency
  - Parent continues after the *Fork* operation
  - Child begins operation separate from the parent
  - *Fork* creates concurrency

- **Join** is called by both the parent and child
  - Child calls *Join* after it finishes (implicitly on exit)
  - Parent waits until child joins (continues afterwards)
  - *Join* removes concurrency because child exits
Fork-Join Pattern

- Control flow **divides** (forks) into multiple flows, then **combines** (joins) later
- During a fork, one flow of control becomes two
- Separate flows are “independent”
  - Does “independent” mean “not dependent”?
  - No, it just means that the 2 flows of control “are not constrained to do similar computation”
- During a join, two flows become one, and only this one flow continues
More to the OpenMP Fork-Join Story

- OpenMP uses a fork-join model of parallel execution as a fundamental basis of the language

- All OpenMP programs begin as a single process
  - *Master* thread executes until a parallel region is encountered

- OpenMP runtime systems executes the parallel region by forking a team of *(Worker)* parallel threads
  - Statements in parallel region are executed by worker threads

- Team threads join with master at parallel region end
Loop-level Parallelization Paradigm

- Execute each loop in parallel
  - Where possible
- Easy to parallelize code
- Incremental parallelization
  - One loop at a time
  - What happens between loops?
- Fine-grain overhead
  - Frequent synchronization
- Performance determined by sequential part (Why?)

```c
C$OMP PARALLEL DO
do i=1,n
  ..........
enddo
alpha = xnorm/sum
C$OMP PARALLEL DO
do i=1,n
  ..........
enddo
C$OMP PARALLEL DO
do i=1,n
  ..........
enddo
```
OpenMP Fork-Join Summary

- OpenMP parallelism is Fork-Join parallelism
- Parallel regions have logical Fork-Join semantics
  - OMP runtime implements a Fork-Join execution model
  - Parallel regions can be nested!!!
    - can create arbitrary Fork-Join structures
- OpenMP tasks are an explicit Fork-Join construct
Load Balancing

- TBB and Cilk Plus use **work stealing** to automatically balance fork-join work
- In a work-stealing scheduler, each thread is a **worker**
- Each worker maintains a stack of tasks
- When a worker’s stack is empty, it grabs from the **bottom** of another random worker
  - Tasks at the bottom of a stack are from the beginning of the call tree – tend to be a bigger piece of work
  - Stolen work will be distant from stack’s owner, minimizing cache conflicts
Pipeline

- A pipeline is a linear sequence of stages
- Data flows through the pipeline
  - From Stage 1 to the last stage
  - Each stage performs some task
    - uses the result from the previous stage
  - Data is thought of as being composed of units (items)
  - Each data unit can be processed separately in pipeline
- Pipeline computation is a special form of *producer-consumer* parallelism
  - Producer tasks output data …
  - … used as input by consumer tasks
Parallel Pipeline Pattern

• Advantages:
  – Conceptually simple
  – Allows for modularity
  – Parallelizes as much as possible, even when some stages are serial, by overlapping
  – Accommodates I/O as serial stages
Parallel Pipeline Pattern

- Disadvantages:
  - Serial computation is still a bottleneck
  - Somewhat difficult to implement well from scratch

![Diagram of pipeline pattern with serial and parallel stages]
Implementation Strategies

- Stage-bound workers
  - Each stage has a number of workers
    - Serial stages have only one
  - Each worker takes a waiting item, performs work, then passes item to the next stage

- Essentially the same as map
  - Simple, but no data locality for each item
**Stage-Bound Workers**

First, each worker grabs input and begins processing it.

Suppose this one finishes first.

The item gets passed to the serial stage.

Since it’s out of order, it must wait to be processed.

Meanwhile, the finished worker grabs more input.

The serial stage accepts the first item.

Now that the first item is processed, the second one can enter the serial stage.
Implementation Strategies

- Item-bound workers
  - Each worker handles an item at a time
  - Worker is responsible for item through whole pipeline
  - On finishing last stage, loops back to beginning for next item
  - More complex, but has much better data locality for items
    - Each item has a better chance of remaining in cache throughout pipeline
  - Workers can get stuck waiting at serial stages
**Item-Bound Workers**

Each worker gets an item, which it carries through the pipeline.

If an item arrives at a serial stage in order, the worker continues.

Otherwise, it must block until its turn comes.

When an item reaches the end, its worker starts over at the first stage.
**Implementation Strategies**

- Hybrid (as implemented in TBB)
  - Workers begin as item-bound
  - When entering a serial stage, the worker checks whether it’s ready to process the item now
    - If so, the worker continues into the stage
    - Otherwise, it *parks* the item, leaving it for another worker, and starts over
  - When leaving a serial stage, the worker checks for a parked item, spawning a new worker to handle it
  - Retains good data locality without requiring workers to block at serial stages
  - No locks needed; works with greedy schedulers
Hybrid Workers

Each worker gets an item, which it intends to carry through the pipeline.

If an item arrives at a serial stage in order, its worker continues.

Otherwise, the worker “parks” the item and abandons it …

… starting over at the first stage.

Whenever an item finishes a serial stage, it checks for a parked item.

If there is one, a new worker is spawned to go through the rest of the pipeline.

When a worker finishes, it starts over at the first stage.
Methodological Design

- **Partition**
  - Task/data decomposition

- **Communication**
  - Task execution coordination

- **Agglomeration**
  - Evaluation of the structure

- **Mapping**
  - Resource assignment

---

I. Foster, “Designing and Building Parallel Programs,” Addison-Wesley, 1995. Book is online, see webpage.
Partitioning

- Partitioning stage is intended to expose opportunities for parallel execution
- Focus on defining large number of small task to yield a fine-grained decomposition of the problem
- A good partition divides into small pieces both the computational tasks associated with a problem and the data on which the tasks operates
- *Domain decomposition* focuses on computation data
- *Functional decomposition* focuses on computation tasks
- Mixing domain/functional decomposition is possible
Communication (Interaction)

- Tasks generated by a partition must interact to allow the computation to proceed
  - Information flow: data and control

- Types of communication
  - *Local* vs. *Global*: locality of communication
  - *Structured* vs. *Unstructured*: communication patterns
  - *Static* vs. *Dynamic*: determined by runtime conditions
  - *Synchronous* vs. *Asynchronous*: coordination degree

- Granularity and frequency of communication
  - Size of data exchange

- Think of communication as interaction and control
  - Applicable to both shared and distributed memory parallelism
Agglomeration

- Move from parallel abstractions to real implementation
- Revisit partitioning and communication
  - View to efficient algorithm execution
- Is it useful to *agglomerate*?
  - What happens when tasks are combined?
- Is it useful to *replicate* data and/or computation?
- Changes important algorithm and performance ratios
  - *Surface-to-volume*: reduction in communication at the expense of decreasing parallelism
  - *Communication/computation*: which cost dominates
- Replication may allow reduction in communication
- Maintain flexibility to allow overlap
Mapping

- Specify where each task is to execute
  - Less of a concern on shared-memory systems
- Attempt to minimize execution time
  - Place concurrent tasks on different processors to enhance physical concurrency
  - Place communicating tasks on same processor, or on processors close to each other, to increase locality
  - Strategies can conflict!
- Mapping problem is \textit{NP-complete}
  - Use problem classifications and heuristics
- Static and dynamic load balancing
Types of Parallel Programs

- Flavors of parallelism
  - Data parallelism
    - all processors do same thing on different data
  - Task parallelism
    - processors are assigned tasks that do different things

- Parallel execution models
  - Data parallel
  - Pipelining (Producer-Consumer)
  - Task graph
  - Work pool
  - Master-Worker
Computations in any parallel algorithms can be viewed as a task dependency graph.

Task dependency graphs can be non-trivial:
- Pipeline
- Arbitrary (represents the algorithm dependencies)

Numbers are time taken to perform task.
Task Graph Performance

- Determined by the *critical path (span)*
  - Sequence of dependent tasks that takes the longest time

- Critical path length bounds parallel execution time

Min time = 27

Min time = 34
Task Assignment (Mapping) to Processors

- Given a set of tasks and number of processors
- How to assign tasks to processors?
- Should take dependencies into account
- Task mapping will determine execution time

(a) Total time = ?

(b) Total time = ?
Bag o’ Tasks Model and Worker Pool

- Set of tasks to be performed
- How do we schedule them?
  - Find independent tasks
  - Assign tasks to available processors
- Bag o’ Tasks approach
  - Tasks are stored in a bag waiting to run
  - If all dependencies are satisfied, it is moved to a ready to run queue
  - Scheduler assigns a task to a free processor
- Dynamic approach that is effective for load balancing
Master-Worker Parallelism

- One or more master processes generate work
- Masters allocate work to worker processes
- Workers idle if have nothing to do
- Workers are mostly stupid and must be told what to do
  - Execute independently
  - May need to synchronize, but most be told to do so
- Master may become the bottleneck if not careful
- What are the performance factors and expected performance behavior
  - Consider task granularity and asynchrony
  - How do they interact?