Midterm Review

Parallel Computing
CIS 410/510
Department of Computer and Information Science

UNIVERSITY OF OREGON
Lecture 1
Introduction
Outline

- Course Overview
  - What is CIS 410/510?
  - What is expected of you?
  - What will you learn in CIS 410/510?

- Parallel Computing
  - What is it?
  - What motivates it?
  - Trends that shape the field
  - Large-scale problems and high-performance
  - Parallel architecture types
  - Scalable parallel computing and performance
Overview

- Broad/Old field of computer science concerned with:
  - Architecture, HW/SW systems, languages, programming paradigms, algorithms, and theoretical models
  - Computing in parallel

- Performance is the *raison d’être* for parallelism
  - High-performance computing
  - Drives computational science revolution

- Topics of study
  - Parallel architectures
  - Parallel programming
  - Parallel algorithms
  - Parallel performance models and tools
  - Parallel applications
Parallel Processing – What is it?

- A parallel computer is a computer system that uses multiple processing elements simultaneously in a cooperative manner to solve a computational problem.
- Parallel processing includes techniques and technologies that make it possible to compute in parallel:
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …
- Parallel computing is an evolution of serial computing:
  - Parallelism is natural
  - Computing problems differ in level / type of parallelism
- Parallelism is all about performance! Really?
Concurrent Review

- Consider multiple tasks to be executed in a computer
- Tasks are concurrent with respect to each if
  - They can execute at the same time (*concurrent execution*)
  - Implies that there are no dependencies between the tasks
- Dependencies
  - If a task requires results produced by other tasks in order to execute correctly, the task’s execution is *dependent*
  - If two tasks are dependent, they are not concurrent
  - Some form of synchronization must be used to enforce (satisfy) dependencies
- Concurrency is fundamental to computer science
  - Operating systems, databases, networking, …
Concurrency and Parallelism

- Concurrent is not the same as parallel! Why?
- Parallel execution
  - Concurrent tasks *actually* execute at the same time
  - Multiple (processing) resources *have* to be available
- **Parallelism = concurrency + “parallel” hardware**
  - Both are required
  - Find concurrent execution opportunities
  - Develop application to execute in parallel
  - Run application on parallel hardware
- Is a parallel application a concurrent application?
- Is a parallel application run with one processor parallel? Why or why not?
Parallelism

- There are granularities of parallelism (parallel execution) in programs
  - Processes, threads, routines, statements, instructions, …
  - Think about what are the software elements that execute concurrently
- These must be supported by hardware resources
  - Processors, cores, … (execution of instructions)
  - Memory, DMA, networks, … (other associated operations)
  - All aspects of computer architecture offer opportunities for parallel hardware execution
- Concurrency is a necessary condition for parallelism
  - Where can you find concurrency?
  - How is concurrency expressed to exploit parallel systems?
Why use parallel processing?

- Two primary reasons (both performance related)
  - Faster time to solution (response time)
  - Solve bigger computing problems (in same time)
- Other factors motivate parallel processing
  - Effective use of machine resources
  - Cost efficiencies
  - Overcoming memory constraints
- Serial machines have inherent limitations
  - Processor speed, memory bottlenecks, …
- Parallelism has become the future of computing
- Performance is still the driving concern
- Parallelism = concurrency + parallel HW + performance
Perspectives on Parallel Processing

- Parallel computer architecture
  - Hardware needed for parallel execution?
  - Computer system design
- (Parallel) Operating system
  - How to manage systems aspects in a parallel computer
- Parallel programming
  - Libraries (low-level, high-level)
  - Languages
  - Software development environments
- Parallel algorithms
- Parallel performance evaluation
- Parallel tools
  - Performance, analytics, visualization, …
Why study parallel computing today?

- Computing architecture
  - Innovations often drive to novel programming models
- Technological convergence
  - The “killer micro” is ubiquitous
  - Laptops and supercomputers are fundamentally similar!
  - Trends cause diverse approaches to converge
- Technological trends make parallel computing inevitable
  - Multi-core processors are here to stay!
  - Practically every computing system is operating in parallel
- Understand fundamental principles and design tradeoffs
  - Programming, systems support, communication, memory, …
  - Performance
- Parallelism is the future of computing
Inevitability of Parallel Computing

- Application demands
  - Insatiable need for computing cycles
- Technology trends
  - Processor and memory
- Architecture trends
- Economics
- Current trends:
  - Today’s microprocessors have multiprocessor support
  - Servers and workstations available as multiprocessors
  - Tomorrow’s microprocessors are multiprocessors
  - Multi-core is here to stay and #cores/processor is growing
  - Accelerators (GPUs, gaming systems)
Broad Parallel Architecture Issues

- Resource allocation
  - How many processing elements?
  - How powerful are the elements?
  - How much memory?

- Data access, communication, and synchronization
  - How do the elements cooperate and communicate?
  - How are data transmitted between processors?
  - What are the abstractions and primitives for cooperation?

- Performance and scalability
  - How does it all translate into performance?
  - How does it scale?
Leveraging Moore’s Law

- More transistors = more parallelism opportunities
- Microprocessors
  - Implicit parallelism
    - pipelining
    - multiple functional units
    - superscalar
  - Explicit parallelism
    - SIMD instructions
    - long instruction works
What’s Driving Parallel Computing Architecture?

Processor-DRAM Memory Gap (latency)

"Moore’s Law"

Processor-Memory Performance Gap: (grows 50% / year)

von Neumann bottleneck!!

(memory wall)

CIS 410/510: Parallel Computing, University of Oregon, Spring 2014 Midterm Review
Microprocessor Transistor Counts (1971-2011)

Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanović
Slide from Kathy Yelick
What’s Driving Parallel Computing Architecture?

Power is the root cause of all this

A hardware issue just became a software problem

power wall
Classifying Parallel Systems – Flynn’s Taxonomy

- Distinguishes multi-processor computer architectures along the two independent dimensions
  - Instruction and Data
  - Each dimension can have one state: Single or Multiple
- SISD: Single Instruction, Single Data
  - Serial (non-parallel) machine
- SIMD: Single Instruction, Multiple Data
  - Processor arrays and vector machines
- MISD: Multiple Instruction, Single Data (weird)
- MIMD: Multiple Instruction, Multiple Data
  - Most common parallel computer systems
Parallel Architecture Types

- **Instruction-Level Parallelism**
  - Parallelism captured in instruction processing

- **Vector processors**
  - Operations on multiple data stored in vector registers

- **Shared-memory Multiprocessor (SMP)**
  - Multiple processors sharing memory
  - Symmetric Multiprocessor (SMP)

- **Multicomputer**
  - Multiple computer connect via network
  - Distributed-memory cluster

- **Massively Parallel Processor (MPP)**
Scalability

- A program can scale up to use many processors
  - What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation
  - If double the number of processors, what to expect?
  - Is scalability linear?
- Use parallel efficiency measure
  - Is efficiency retained as problem size increases?
- Apply performance metrics
Top 500 Benchmarking Methodology

- Listing of the world’s 500 most powerful computers
- Yardstick for high-performance computing (HPC)
  - $R_{\text{max}}$: maximal performance Linpack benchmark
    - dense linear system of equations ($Ax = b$)
- Data listed
  - $R_{\text{peak}}$: theoretical peak performance
  - $N_{\text{max}}$: problem size needed to achieve $R_{\text{max}}$
  - $N_{1/2}$: problem size needed to achieve 1/2 of $R_{\text{max}}$
  - Manufacturer and computer type
  - Installation site, location, and year
- Updated twice a year at SC and ISC conferences
Major Changes to Software and Algorithms

- What were we concerned about before and now?
- Must rethink the design for exascale
  - Data movement is expensive (Why?)
  - Flops per second are cheap (Why?)
- Need to reduce communication and synchronisation
- Need to develop fault-resilient algorithms
- How do we deal with massive parallelism?
- Software must adapt to the hardware (autotuning)
Scalable Parallel Computing

- Scalability in parallel architecture
  - Processor numbers
  - Memory architecture
  - Interconnection network
  - Avoid critical architecture bottlenecks

- Scalability in computational problem
  - Problem size
  - Computational algorithms
    - Computation to memory access ratio
    - Computation to communication ration

- Parallel programming models and tools
- Performance scalability
Lecture 2
Parallel Computer Architecture
Parallel Architecture Types

• Uniprocessor
  – Scalar processor
  
  processor
  memory
  
  – Vector processor

  processor
  vector
  memory

  – Single Instruction Multiple Data (SIMD)

  processor
  memory

  Multiprocessor (SMP)
  – Shared memory address space
  – Bus-based memory system

  processor
  ... 
  processor
  bus
  memory

  – Interconnection network

  processor
  ... 
  processor
  network
  memory
Parallel Architecture Types (2)

• Distributed Memory Multiprocessor
  – Message passing between nodes

  • Massively Parallel Processor (MPP)
    • Many, many processors

• Cluster of SMPs
  – Shared memory addressing within SMP node
  – Message passing between SMP nodes

  – Can also be regarded as MPP if processor number is large
Parallel Architecture Types (3)

- Multicore
  - Multicore processor
    - Cores can be hardware multithreaded (hyperthread)
  - GPU accelerator
  - “Fused” processor accelerator

- Multicore SMP+GPU Cluster
  - Shared memory addressing within SMP node
  - Message passing between SMP nodes
  - GPU accelerators attached
How do you get parallelism in the hardware?

- Instruction-Level Parallelism (ILP)
- Data parallelism
  - Increase amount of data to be operated on at same time
- Processor parallelism
  - Increase number of processors
- Memory system parallelism
  - Increase number of memory units
  - Increase bandwidth to memory
- Communication parallelism
  - Increase amount of interconnection between elements
  - Increase communication bandwidth
Vector Processing

- Scalar processing
  - Processor instructions operate on scalar values
  - integer registers and floating point registers
- Vectors
  - Set of scalar data
  - Vector registers
    - integer, floating point (typically)
  - Vector instructions operate on vector registers (SIMD)
- Vector unit pipelining
- Multiple vector units
- Vector chaining
Data Parallel Architectures

- SIMD (Single Instruction Multiple Data)
  - Logical single thread (instruction) of control
  - Processor associated with data elements

- Architecture
  - Array of simple processors with memory
  - Processors arranged in a regular topology
  - Control processor issues instructions
    - All processors execute same instruction (maybe disabled)
  - Specialized synchronization and communication
  - Specialized reduction operations
  - Array processing
Shared Physical Memory

- Add processors to single processor computer system
- Processors share computer system resources
  - Memory, storage, …
- Sharing physical memory
  - Any processor can reference any memory location
  - Any I/O controller can reference any memory address
  - Single physical memory address space
- Operating system runs on any processor, or all
  - OS see single memory address space
  - Uses shared memory to coordinate
- Communication occurs as a result of loads and stores
Caching in Shared Memory Systems

- Reduce average latency
  - automatic replication closer to processor
- Reduce average bandwidth
- Data is logically transferred from producer to consumer to memory
  - store reg → mem
  - load reg ← mem
- Processors can share data efficiently
- What happens when store and load are executed on different processors?
- Cache coherence problems
Shared Memory Multiprocessors (SMP)

- **Architecture types**

  Single processor
  - P
  - M

  Multiple processors
  - P
  - M
  - multi-port

- **Differences lie in memory system interconnection**

  Interconnect:
  - Processor
  - Mem
  - I/O ctrl
  - I/O devices

  What does this look like?

CIS 410/510: Parallel Computing, University of Oregon, Spring 2014  Midterm Review
Bus-based SMP

- Memory bus handles all memory read/write traffic
- Processors share bus
- **Uniform Memory Access (UMA)**
  - Memory (not cache) uniformly equidistant
  - Take same amount of time (generally) to complete
- May have multiple memory modules
  - Interleaving of physical address space
- Caches introduce memory hierarchy
  - Lead to data consistency problems
  - Cache coherency hardware necessary (**CC-UMA**)
**Crossbar SMP**

- Replicates memory bus for every processor and I/O controller
  - Every processor has direct path
- UMA SMP architecture
- Can still have cache coherency issues
- Multi-bank memory or interleaved memory

**Advantages**
- Bandwidth scales linearly (no shared links)

**Problems**
- High incremental cost (cannot afford for many processors)
- Use switched multi-stage interconnection network
“Dance Hall” SMP and Shared Cache

- Interconnection network connects processors to memory
- Centralized memory (UMA)
- Network determines performance
  - Continuum from bus to crossbar
  - Scalable memory bandwidth
- Memory is physically separated from processors
- Could have cache coherence problems
- Shared cache reduces coherence problem and provides fine grained data sharing
Natural Extensions of the Memory System

- **Shared Cache**
- **Centralized Memory**
  - Dance Hall, UMA
- **Crossbar, Interleaved**
- **Distributed Memory (NUMA)**
Non-Uniform Memory Access (NUMA) SMPs

- Distributed memory
- Memory is physically resident close to each processor
- Memory is still shared
- **Non-Uniform Memory Access (NUMA)**
  - Local memory and remote memory
  - Access to local memory is faster, remote memory slower
  - Access is non-uniform
  - Performance will depend on data locality
- Cache coherency is still an issue (more serious)
- Interconnection network architecture is more scalable
Cache Coherency and SMPs

- Caches play key role in SMP performance
  - Reduce average data access time
  - Reduce bandwidth demands placed on shared interconnect

- Private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
    - they’ll keep accessing stale value in their caches

  ⇒ Cache coherency problem

- What do we do about it?
  - Organize the memory hierarchy to make it go away
  - Detect and take actions to eliminate the problem
Definitions

- Memory operation (load, store, read-modify-write, …)
- Memory issue is operation presented to memory system

Processor perspective
- Write: subsequent reads return the value
- Read: subsequent writes cannot affect the value

Coherent memory system
- There exists a serial order of memory operations on each location such that
  - operations issued by a process appear in order issued
  - value returned by each read is that written by previous write
⇒ write propagation + write serialization
Memory Consistency

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to each other
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it

- Implications for both programmer and system designer
  - Programmer uses to reason about correctness
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware

- Contract between programmer and system
Sequential Consistency

- Total order achieved by interleaving accesses from different processes
  - Maintains *program order*
  - Memory operations (from all processes) appear to issue, execute, and complete atomically with respect to others
  - As if there was a single memory (no cache)

“A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” [Lamport, 1979]
Sequential Consistency (Sufficient Conditions)

- There exist a total order consistent with the memory operations becoming visible in program order

- Sufficient Conditions
  - every process issues memory operations in program order
  - after write operation is issued, the issuing process waits for write to complete before issuing next memory operation (atomic writes)
  - after a read is issued, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next memory operation

- Cache-coherent architectures implement consistency
Memory Consistency

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to each other
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
- Implications for both programmer and system designer
  - Programmer uses to reason about correctness
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system
- Need coherency systems to enforce memory consistency
Context for Scalable Cache Coherence

Realizing programming models through net transaction protocols
- efficient node-to-net interface
- interprets transactions

Scalable Networks
- many simultaneous transactions

Scalable distributed memory

Caches naturally replicate data
- coherence through bus
- snooping protocols
- consistency

Need cache coherence protocols that scale!
- no broadcast or single point of order
Distributed Memory Multiprocessors

- Each processor has a local memory
  - Physically separated memory address space

- Processors must communicate to access non-local data
  - Message communication (message passing)
    - *Message passing architecture*
  - Processor interconnection network

- Parallel applications must be partitioned across
  - Processors: execution units
  - Memory: data partitioning

- Scalable architecture
  - Small incremental cost to add hardware (cost of node)
Distributed Memory (MP) Architecture

- Nodes are complete computer systems
  - Including I/O
- Nodes communicate via interconnection network
  - Standard networks
  - Specialized networks
- Network interfaces
  - Communication integration
- Easier to build
Network Performance Measures

Overhead: latency of interface vs. Latency: network
Performance Metrics: Latency and Bandwidth

- **Bandwidth**
  - Need high bandwidth in communication
  - Match limits in network, memory, and processor
  - Network interface speed vs. network bisection bandwidth

- **Latency**
  - Performance affected since processor may have to wait
  - Harder to overlap communication and computation
  - Overhead to communicate is a problem in many machines

- **Latency hiding**
  - Increases programming system burden
  - Examples: communication/computation overlaps, prefetch
Scalable, High-Performance Interconnect

- Interconnection network is core of parallel architecture
- Requirements and tradeoffs at many levels
  - Elegant mathematical structure
  - Deep relationship to algorithm structure
  - Hardware design sophistication
- Little consensus
  - Performance metrics?
  - Cost metrics?
  - Workload?
  - …
What Characterizes an Interconnection Network?

- **Topology** (what)
  - Interconnection structure of the network graph

- **Routing Algorithm** (which)
  - Restricts the set of paths that messages may follow
  - Many algorithms with different properties

- **Switching Strategy** (how)
  - How data in a message traverses a route
  - *circuit switching* vs. *packet switching*

- **Flow Control Mechanism** (when)
  - When a message or portions of it traverse a route
  - What happens when traffic is encountered?
Advantages of Shared Memory Architectures

- Compatibility with SMP hardware
- Ease of programming when communication patterns are complex or vary dynamically during execution
- Ability to develop applications using familiar SMP model, attention only on performance critical accesses
- Lower communication overhead, better use of BW for small items, due to implicit communication and memory mapping to implement protection in hardware, rather than through I/O system
- HW-controlled caching to reduce remote communication by caching of all data, both shared and private
**Advantages of Distributed Memory Architectures**

- The hardware can be simpler (especially versus NUMA) and is more scalable.
- Communication is explicit and simpler to understand.
- Explicit communication focuses attention on costly aspect of parallel computation.
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization.
- Easier to use sender-initiated communication, which may have some advantages in performance.
Clusters of SMPs

- Clustering
  - Integrated packaging of nodes

- Motivation
  - Ammortize node costs by sharing packaging and resources
  - Reduce network costs
  - Reduce communications bandwidth requirements
  - Reduce overall latency
  - More parallelism in a smaller space
  - Increase node performance

- Scalable parallel systems today are built as SMP clusters
Lecture 3 and 4
Parallel Performance Theory
What is Parallel Performance?

- Here we are concerned with performance issues when using a parallel computing environment
  - Performance with respect to parallel computation
- Performance is the *raison d’être* for parallelism
  - Parallel performance versus sequential performance
  - If the “performance” is not better, parallelism is not necessary
- *Parallel processing* includes techniques and technologies necessary to compute in parallel
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …
- Parallelism must deliver performance
  - How? How well?
Performance Expectation (Loss)

- If each processor is rated at $k$ MFLOPS and there are $p$ processors, should we see $k*p$ MFLOPS performance?
- If it takes 100 seconds on 1 processor, shouldn’t it take 10 seconds on 10 processors?
- Several causes affect performance
  - Each must be understood separately
  - But they interact with each other in complex ways
    - Solution to one problem may create another
    - One problem may mask another
- Scaling (system, problem size) can change conditions
- Need to understand performance space
Embarrassingly Parallel Computations

- An embarrassingly parallel computation is one that can be obviously divided into completely independent parts that can be executed simultaneously
  - In a truly embarrassingly parallel computation there is no interaction between separate processes
  - In a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way
- Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms
  - If it takes $T$ time sequentially, there is the potential to achieve $T/P$ time running in parallel with $P$ processors
  - What would cause this not to be the case always?
Scalability

- A program can scale up to use many processors
  - What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation
  - If double the number of processors, what to expect?
  - Is scalability linear?
- Use parallel efficiency measure
  - Is efficiency retained as problem size increases?
- Apply performance metrics
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
  \[ \text{Efficiency} = \frac{S_p}{p} \]
- $\text{Cost}(p)$ ($C_p$) is the cost
  \[ \text{Cost} = p \times T_p \]
- 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 $p \to \infty$
  \[ S_p = \frac{1}{f} \]
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?
  - \( S_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par} \)
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

serial work

parallelizable work

Time

P=1
P=2
P=4
P=8
Amdahl versus Gustafson-Baris

Gustafson-Baris

P=1  P=2  P=4  P=8

serial work

parallelizable work

Time
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
**Work-Span Example**

- Let each task take 1 unit of time
- DAG at the right has 7 tasks
- $T_1 = 7$
  - All tasks have to be executed
  - Tasks are executed in a serial order
  - Can the execute in any order?
- $T_\infty = 5$
  - Time along the *critical path*
  - In this case, it is the longest pathlength of any task order that maintains necessary dependencies
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
  - $T_P \leq (T_1 - T_\infty) / P + T_\infty$
Consider Brent’s Lemma for 2 Processors

- $T_1 = 7$
- $T_\infty = 5$
- $T_2 \leq (T_1 - T_\infty) / P + T_\infty$
  \[ \leq (7 - 5) / 2 + 5 \]
  \[ \leq 6 \]
Amdahl was an optimist!
Estimating Running Time

- Scalability requires that $T_\infty$ be dominated by $T_1$

$$T_P \approx T_1 / P + T_\infty \quad \text{if} \quad T_\infty << T_1$$

- Increasing work hurts parallel execution proportionately

- The span impacts scalability, even for finite $P$
Parallel Slack

- Sufficient parallelism implies linear speedup

\[ T_p \approx \frac{T_1}{P} \quad \text{if} \quad \frac{T_1}{T_\infty} \gg P \]

Linear speedup \hspace{1cm} Parallel slack
Scalable Parallel Computing

- Scalability in parallel architecture
  - Processor numbers
  - Memory architecture
  - Interconnection network
  - Avoid critical architecture bottlenecks
- Scalability in computational problem
  - Problem size
  - Computational algorithms
    - Computation to memory access ratio
    - Computation to communication ratio
- Parallel programming models and tools
- Performance scalability
Why Aren’t Parallel Applications Scalable?

- Sequential performance
- Critical Paths
  - Dependencies between computations spread across processors
- Bottlenecks
  - One processor holds things up
- Algorithmic overhead
  - Some things just take more effort to do in parallel
- Communication overhead
  - Spending increasing proportion of time on communication
- Load Imbalance
  - Makes all processor wait for the “slowest” one
  - Dynamic behavior
- Speculative loss
  - Do A and B in parallel, but B is ultimately not needed
Critical Paths

- Long chain of dependence
  - Main limitation on performance
  - Resistance to performance improvement

- Diagnostic
  - Performance stagnates to a (relatively) fixed value
  - Critical path analysis

- Solution
  - Eliminate long chains if possible
  - Shorten chains by removing work from critical path
Bottlenecks

- How to detect?
  - One processor A is busy while others wait
  - Data dependency on the result produced by A

- Typical situations:
  - N-to-1 reduction / computation / 1-to-N broadcast
  - One processor assigning job in response to requests

- Solution techniques:
  - More efficient communication
  - Hierarchical schemes for master slave

- Program may not show ill effects for a long time
- Shows up when scaling
Algorithmic Overhead

- Different sequential algorithms to solve the same problem
- All parallel algorithms are sequential when run on 1 processor
- All parallel algorithms introduce addition operations (Why?)
  - Parallel overhead
- Where should be the starting point for a parallel algorithm?
  - Best sequential algorithm might not parallelize at all
  - Or, it doesn’t parallelize well (e.g., not scalable)
- What to do?
  - Choose algorithmic variants that minimize overhead
  - Use two level algorithms
- Performance is the rub
  - Are you achieving better parallel performance?
  - Must compare with the best sequential algorithm
What is the maximum parallelism possible?

- Depends on application, algorithm, program
  - Data dependencies in execution

- Remember MaxPar
  - Analyzes the earliest possible “time” any data can be computed
  - Assumes a simple model for time it takes to execute instruction or go to memory
  - Result is the maximum parallelism available

- Parallelism varies!
Embarrassingly Parallel Computations

- No or very little communication between processes
- Each process can do its tasks without any interaction with other processes

Examples
- Numerical integration
- Mandelbrot set
- Monte Carlo methods
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*
  \[ \text{Efficiency} = \frac{S_p}{p} \]
- $Cost(p)$ ($C_p$) is the *cost*
  \[ Cost = p \times T_p \]
- Parallel algorithm is *cost-optimal*
  - Parallel time = sequential time ($C_p = T_1, E_p = 100\%$)
Analytical / Theoretical Techniques

- Involves simple algebraic formulas and ratios
  - Typical variables are:
    - data size \( (N) \), number of processors \( (P) \), machine constants
  - Want to model performance of individual operations, components, algorithms in terms of the above
    - be careful to characterize variations across processors
    - model them with max operators
  - Constants are important in practice
    - Use asymptotic analysis carefully

- Scalability analysis
  - Isoefficiency (Kumar)
Isoefficiency

- Goal is to quantify scalability
- How much increase in problem size is needed to retain the same efficiency on a larger machine?
- Efficiency
  - $T_1 / (p \times T_p)$
  - $T_p = \text{computation} + \text{communication} + \text{idle}$
- Isoefficiency
  - Equation for equal-efficiency curves
  - If no solution
    - problem is not scalable in the sense defined by isoefﬁciency
- See original paper by Kumar on webpage
Scalability of Adding $n$ Numbers

- Scalability of a parallel system is a measure of its capacity to increase speedup with more processors.
- Adding $n$ numbers on $p$ processors with strip partition:

$$T_{par} = \frac{n}{p} - 1 + 2 \log p$$

$$\text{Speedup} = \frac{n - 1}{\frac{n}{p} - 1 + 2 \log p}$$

$$\approx \frac{n}{\frac{n}{p} + 2 \log p}$$

$$\text{Efficiency} = \frac{S}{p} = \frac{n}{n + 2p \log p}$$
Problem Size and Overhead

- Informally, problem size is expressed as a parameter of the input size.
- A consistent definition of the size of the problem is the total number of basic operations ($T_{seq}$).
  - Also refer to problem size as “work ($W = T_{seq}$).”
- Overhead of a parallel system is defined as the part of the cost not in the best serial algorithm.
- Denoted by $T_O$, it is a function of $W$ and $p$.
  \[
  T_O(W,p) = pT_{par} - W \quad (pT_{par} \text{ includes overhead})
  \]
  \[
  T_O(W,p) + W = pT_{par}
  \]
Isoefficiency Function

- With a fixed efficiency, $W$ is as a function of $p$

$$T_{par} = \frac{W + T_o(W, p)}{p}$$

$$W = T_{seq}$$

$$\text{Speedup} = \frac{W}{T_{par}} = \frac{Wp}{W + T_o(W, p)}$$

$$\text{Efficiency} = \frac{S}{p} = \frac{W}{W + T_o(W, p)} = \frac{1}{1 + \frac{T_o(W, p)}{W}}$$

$$E = \frac{1}{1 + \frac{T_o(W, p)}{W}} \rightarrow \frac{T_o(W, p)}{W} = \frac{1 - E}{E}$$

$$W = \frac{E}{1 - E} T_o(W, p) = KT_o(W, p) \quad \text{Isoefficiency Function}$$
Isoefficiency Function of Adding n Numbers

- Overhead function:
  \[ T_0(W,p) = pT_{par} - W = 2p \log(p) \]

- Isoefficiency function:
  \[ W = K \cdot 2p \log(p) \]

- If \( p \) doubles, \( W \) needs also to be doubled to roughly maintain the same efficiency

- Isoefficiency functions can be more difficult to express for more complex algorithms
More Complex Isoefficiency Functions

- A typical overhead function $T_O$ can have several distinct terms of different orders of magnitude with respect to both $p$ and $W$

- We can balance $W$ against each term of $T_O$ and compute the respective isoefficiency functions for individual terms
  - Keep only the term that requires the highest grow rate with respect to $p$
  - This is the asymptotic isoefficiency function
Isoefficiency

- Consider a parallel system with an overhead function

\[ T_o = p^{3/2} + p^{3/4}W^{3/4} \]

- Using only the first term

\[ W = Kp^{3/2} \]

- Using only the second term

\[ W = Kp^{3/4}W^{3/4} \]

\[ W^{1/4} = Kp^{3/4} \]

\[ W = K^4 p^3 \]

- \( K^4 p^3 \) is the overall asymptotic isoefficiency function
**BSP Overview**

- Bulk Synchronous Parallelism
- A parallel programming model
- Invented by Leslie Valiant at Harvard
- Enables performance prediction
- SPMD (Single Program Multiple Data) style
- Supports both direct memory access and message passing semantics
- BSPlib is a BSP library implemented at Oxford
Components of BSP Computer

- A set of processor-memory pairs
- A communication point-to-point network
- A mechanism for efficient barrier synchronization of all processors
BSP Supersteps

- A BSP computation consists of a sequence of supersteps.
- In each superstep, processes execute computations using locally available data, and issue communication requests.
- Processes synchronized at the end of the superstep, at which all communications issued have been completed.
BSP Performance Model Parameters

- $p$ = number of processors
- $l$ = barrier latency, cost of achieving barrier synchronization
- $g$ = communication cost per word
- $s$ = processor speed
- $l$, $g$, and $s$ are measured in FLOPS
- Any processor sends and receives at most $h$ messages in a single superstep (called $h$-relation communication)
- Time for a superstep = max number of local operations performed by any one processor + $g \times h + l$
Parallel Programming Models

- Two general models of parallel program
  - Task parallel
    - problem is broken down into tasks to be performed
    - individual tasks are created and communicate to coordinate operations
  - Data parallel
    - problem is viewed as operations of parallel data
    - data distributed across processes and computed locally

- Characteristics of scalable parallel programs
  - Data domain decomposition to improve data locality
  - Communication and latency do not grow significantly
Shared Memory Parallel Programming

- Shared memory address space
- (Typically) easier to program
  - Implicit communication via (shared) data
  - Explicit synchronization to access data
- Programming methodology
  - Manual
    - multi-threading using standard thread libraries
  - Automatic
    - parallelizing compilers
    - OpenMP parallelism directives
  - Explicit threading (e.g. POSIX threads)
Distributed Memory Parallel Programming

- Distributed memory address space
- (Relatively) harder to program
  - Explicit data distribution
  - Explicit communication via messages
  - Explicit synchronization via messages
- Programming methodology
  - Message passing
    - plenty of libraries to chose from (MPI dominates)
      - send-receive, one-sided, active messages
  - Data parallelism
Lecture 5
Parallel Programming Patterns
Overview and Map Pattern
Parallel Models 101

- Sequential models
  - von Neumann (RAM) model

- Parallel model
  - A parallel computer is simply a collection of *processors interconnected* in some manner to coordinate activities and exchange data
  - Models that can be used as general frameworks for describing and analyzing parallel algorithms
    - *Simplicity*: description, analysis, architecture independence
    - *Implementability*: able to be realized, reflect performance

- Three common parallel models
  - Directed acyclic graphs, shared-memory, network
Directed Acyclic Graphs (DAG)

- Captures data flow parallelism
- Nodes represent operations to be performed
  - Inputs are nodes with no incoming arcs
  - Output are nodes with no outgoing arcs
  - Think of nodes as tasks
- Arcs are paths for flow of data results
- DAG represents the operations of the algorithm and implies precedent constraints on their order

```
for (i=1; i<100; i++)
    a[i] = a[i-1] + 100;
```
Shared Memory Model

- Parallel extension of RAM model (PRAM)
  - Memory size is infinite
  - Number of processors in unbounded
  - Processors communicate via the memory
  - Every processor accesses any memory location in 1 cycle
  - Synchronous
    - All processors execute same algorithm synchronously
      - READ phase
      - COMPUTE phase
      - WRITE phase
    - Some subset of the processors can stay idle
  - Asynchronous
**Network Model**

- $G = (N,E)$
  - $N$ are processing nodes
  - $E$ are bidirectional communication links
- Each processor has its own memory
- No shared memory is available
- Network operation may be synchronous or asynchronous
- Requires communication primitives
  - Send $(X, i)$
  - Receive $(Y, j)$
- Captures message passing model for algorithm design
Parallelism

- Ability to execute different parts of a computation concurrently on different machines
- Why do you want parallelism?
  - Shorter running time or handling more work
- What is being parallelized?
  - Task: instruction, statement, procedure, …
  - Data: data flow, size, replication
  - Parallelism granularity
    - Coarse-grain versus fine-grained
- Thinking about parallelism
- Evaluation
Why is parallel programming important?

- Parallel programming has matured
  - Standard programming models
  - Common machine architectures
  - Programmer can focus on computation and use suitable programming model for implementation
- Increase portability between models and architectures
- Reasonable hope of portability across platforms
- Problem
  - Performance optimization is still platform-dependent
  - Performance portability is a problem
  - Parallel programming methods are still evolving
Parallelism Views

- Where can we find parallelism?
- Program (task) view
  - Statement level
    - Between program statements
    - Which statements can be executed at the same time?
  - Block level / Loop level / Routine level / Process level
    - Larger-grained program statements
- Data view
  - How is data operated on?
  - Where does data reside?
- Resource view
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 may depend on certain resources.
  - For example, due to where data is located.
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: statement 2;

- Case 2:
  - Processor 1: statement 2;
  - Processor 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
  
  \[
  \text{statement1; statement2;}
  \]
  
  must be equivalent to
  
  \[
  \text{statement2; statement1;}
  \]

- 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: \( b = 1 \);
- Example 2
  S1: \( a = 1 \);
  S2: \( b = a \);
- Example 3
  S1: \( a = f(x) \);
  S2: \( a = b \);
- Example 4
  S1: \( a = b \);
  S2: \( b = 1 \);

- Statements are independent
- Dependent (true (flow) dependence)
  - Second is dependent on first
  - Can you remove dependency?
- Dependent (output dependence)
  - Second is dependent on first
  - Can you remove dependency? How?
- Dependent (anti-dependence)
  - First is dependent on second
  - Can you remove dependency? How?
True Dependence and Anti-Dependence

- Given statements $S_1$ and $S_2$,
  
  $S_1$;  
  $S_2$;

- $S_2$ has a **true (flow) dependence** on $S_1$ if and only if $S_2$ reads a value written by $S_1$

- $S_2$ has a **anti-dependence** on $S_1$ if and only if $S_2$ writes a value read by $S_1$
Output Dependence

- Given statements S1 and S2,
  
  S1;
  
  S2;

- S2 has an output dependence on S1 if and only if S2 writes a variable written by S1

- Anti- and output dependences are “name” dependencies
  - Are they “true” dependences?

- How can you get rid of output dependences?
  - Are there cases where you can not?
Statement Dependency Graphs

- Can use graphs to show dependence relationships
- Example
  - S1: \( a = 1 \);
  - S2: \( b = a \);
  - S3: \( a = b + 1 \);
  - S4: \( c = a \);

- \( S_2 \xrightarrow{\text{flow}} S_3 \): \( S_3 \) is flow-dependent on \( S_2 \)
- \( S_1 \xrightarrow{\text{output}} S_3 \): \( S_3 \) is output-dependent on \( S_1 \)
- \( S_2 \xrightarrow{\text{anti}} S_3 \): \( S_3 \) is anti-dependent on \( S_2 \)
When can two statements execute in parallel?

- Statements $S_1$ and $S_2$ can execute in parallel if and only if there are no dependences between $S_1$ and $S_2$
  - True dependences
  - Anti-dependences
  - Output dependences
- Some dependences can be removed by modifying the program
  - Rearranging statements
  - Eliminating statements
How do you compute dependence?

- Data dependence relations can be found by comparing the IN and OUT sets of each node.
- The IN and OUT sets of a statement $S$ are defined as:
  - $\text{IN}(S)$: set of memory locations (variables) that may be used in $S$
  - $\text{OUT}(S)$: set of memory locations (variables) that may be modified by $S$
- Note that these sets include all memory locations that may be fetched or modified.
- As such, the sets can be conservatively large.
IN / OUT Sets and Computing Dependence

- Assuming that there is a path from $S_1$ to $S_2$, the following shows how to intersect the IN and OUT sets to test for data dependence:

$$\text{out}(S_1) \cap \text{in}(S_2) \neq \emptyset$$  \quad S_1 \delta S_2  \text{ flow dependence}

$$\text{in}(S_1) \cap \text{out}(S_2) \neq \emptyset$$  \quad S_1 \delta^{-1} S_2  \text{ anti-dependence}

$$\text{out}(S_1) \cap \text{out}(S_2) \neq \emptyset$$  \quad S_1 \delta^0 S_2  \text{ output dependence}
**Loop-Level Parallelism**

- Significant parallelism can be identified within loops

```plaintext
for (i=0; i<100; i++)
  S1: a[i] = i;

for (i=0; i<100; i++) {
  S1: a[i] = i;
  S2: b[i] = 2*i;
}
```

- Dependencies? What about \( i \), the loop index?

- **DOALL** loop (a.k.a. *foreach* loop)
  - All iterations are independent of each other
  - All statements be executed in parallel at the same time
    - Is this really true?
Iteration Space

- Unroll loop into separate statements / iterations
- Show dependences between iterations

```c
for (i=0; i<100; i++)
    S1: a[i] = i;
```

```c
for (i=0; i<100; i++) {
    S1: a[i] = i;
    S2: b[i] = 2*i;
}
```
Multi-Loop Parallelism

- Significant parallelism can be identified between loops

```c
for (i=0; i<100; i++) a[i] = i;
for (i=0; i<100; i++) b[i] = i;
```

- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?
Loops with Dependencies

Case 1:
for (i=1; i<100; i++)
    a[i] = a[i-1] + 100;

Case 2:
for (i=5; i<100; i++)
    a[i-5] = a[i] + 100;

- Dependencies?
  - What type?
- Is the Case 1 loop parallelizable?
- Is the Case 2 loop parallelizable?
Dependences Between Iterations

for (i=1; i<100; i++) {
    S1: a[i] = …;
    S2: … = a[i-1];
}

- Dependencies?
  - Between a[i] and a[i-1]
- Is parallelism possible?
  - Statements can be executed in “pipeline” manner
Key Ideas for Dependency Analysis

- To execute in parallel:
  - Statement order must not matter
  - Statements must not have dependences
- Some dependences can be removed
- Some dependences may not be obvious
Parallel Patterns

- **Parallel Patterns**: A recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.

- Patterns provide us with a “vocabulary” for algorithm design

- It can be useful to compare parallel patterns with serial patterns

- Patterns are universal – they can be used in *any* parallel programming system
Parallel Patterns

- Nesting Pattern
- Serial / Parallel Control Patterns
- Serial / Parallel Data Management Patterns
- Other Patterns
- Programming Model Support for Patterns
Nesting Pattern

- **Nesting** is the ability to hierarchically compose patterns.
- This pattern appears in both serial and parallel algorithms.
- “Pattern diagrams” are used to visually show the pattern idea where each “task block” is a location of general code in an algorithm.
- Each “task block” can in turn be another pattern in the nesting pattern.
Nesting Pattern

Nesting Pattern: A compositional pattern. Nesting allows other patterns to be composed in a hierarchy so that any task block in the above diagram can be replaced with a pattern with the same input/output and dependencies.
Serial Control Patterns

- Structured serial programming is based on these patterns: sequence, selection, iteration, and recursion
- The nesting pattern can also be used to hierarchically compose these four patterns
- Though you should be familiar with these, it’s extra important to understand these patterns when parallelizing serial algorithms based on these patterns
Serial Control Patterns: Sequence

- **Sequence**: Ordered list of tasks that are executed in a specific order

- Assumption – program text ordering will be followed (obvious, but this will be important when parallelized)

```
1. T = f(A);
2. S = g(T);
3. B = h(S);
```

```
1. T = f(A);
2. S = g(A);
3. B = h(S, T);
```
Serial Control Patterns: Selection

- **Selection**: condition $c$ is first evaluated. Either task $a$ or $b$ is executed depending on the true or false result of $c$.

- Assumptions – $a$ and $b$ are never executed before $c$, and only $a$ or $b$ is executed - never both

```c
1 if (c) {
2    a;
3 } else {
4    b;
5 }
```
Serial Control Patterns: Iteration

- **Iteration**: a condition \( c \) is evaluated. If true, \( a \) is evaluated, and then \( c \) is evaluated again. This repeats until \( c \) is false.

- Complication when parallelizing: potential for dependencies to exist between previous iterations.

```java
1   for (i = 0; i < n;
2       a;
3   }
1   while (c) {
2     a;
3   }
```

![Diagram of iteration control flow]
Parallel Control Patterns

- Parallel control patterns extend serial control patterns.
- Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns.
- Parallel control patterns: fork-join, map, stencil, reduction, scan, recurrence.
Parallel Control Patterns: Fork-Join

- **Fork-join**: allows control flow to fork into multiple parallel flows, then rejoin later
- Cilk Plus implements this with `spawn` and `sync`
  - The call tree is a parallel call tree and functions are spawned instead of called
  - Functions that spawn another function call will continue to execute
  - Caller `syncs` with the spawned function to join the two
- A “join” is different than a “barrier”
  - Sync – only one thread continues
  - Barrier – all threads continue
Parallel Control Patterns: Map

- **Map**: performs a function over every element of a collection
- Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection
- The replicated function is referred to as an “elemental function”
Parallel Control Patterns: Stencil

- **Stencil**: Elemental function accesses a set of “neighbors”, stencil is a generalization of map.
- Often combined with iteration – used with iterative solvers or to evolve a system through time.
- Boundary conditions must be handled carefully in the stencil pattern.
- See stencil lecture…
Parallel Control Patterns: Reduction

- **Reduction**: Combines every element in a collection using an associative “combiner function”

- Because of the associativity of the combiner function, different orderings of the reduction are possible

- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR
Parallel Control Patterns: Reduction

Serial Reduction

Parallel Reduction
Serial Data Management Patterns

- Serial programs can manage data in many ways
- Data management deals with how data is allocated, shared, read, written, and copied
- Serial Data Management Patterns: random read and write, stack allocation, heap allocation, objects
Parallel Data Management Patterns

- To avoid things like race conditions, it is critically important to know when data is, and isn’t, potentially shared by multiple parallel workers.
- Some parallel data management patterns help us with data locality.
- Parallel data management patterns: **pack, pipeline, geometric decomposition, gather, and scatter**
**Parallel Data Management Patterns: Pack**

- **Pack** is used to eliminate unused space in a collection.
- Elements marked `false` are discarded, the remaining elements are placed in a contiguous sequence in the same order.
- Useful when used with `map`.
- **Unpack** is the inverse and is used to place elements back in their original locations.
Parallel Data Management Patterns: Pipeline

- **Pipeline** connects tasks in a producer-consumer manner.

- A linear pipeline is the basic pattern idea, but a pipeline in a DAG is also possible.

- Pipelines are most useful when used with other patterns as they can multiply available parallelism.
Parallel Data Management Patterns: Geometric Decomposition

- Geometric Decomposition – arranges data into subcollections
- Overlapping and non-overlapping decompositions are possible
- This pattern doesn’t necessarily move data, it just gives us another view of it
Parallel Data Management Patterns: Gather

- **Gather** reads a collection of data given a collection of indices
- Think of a combination of map and random serial reads
- The output collection shares the same type as the input collection, but it share the same shape as the indices collection

![Diagram showing the relationship between indices and data elements]

[Diagram showing indices and data elements with annotations]
Parallel Data Management Patterns: Scatter

- **Scatter** is the inverse of gather
- A set of input and indices is required, but each element of the input is written to the output at the given index instead of read from the input at the given index
- Race conditions can occur when we have two writes to the same location!
Mapping

- “Do the same thing many times”
  ```python
  foreach i in foo:
      do something
  ```
- Well-known higher order function in languages like ML, Haskell, Scala
  ```latex
  \text{map} : \forall ab. (a \rightarrow b) \text{List}\langle a\rangle \rightarrow \text{List}\langle b\rangle
  ```
  applies a function each element in a list and returns a list of results
Key Idea

- Map is a “foreach loop” where each iteration is independent

Embarrassingly Parallel

Independence is a big win. We can run map completely in parallel. Significant speedups! More precisely: $T(\infty)$ is $O(1)$ plus implementation overhead that is $O(\log n)$…so $T(\infty) \in O(\log n)$. 
for(int n=0;
    n< array.length;
    ++n)
{
    process(array[n]);
}
parallel_for_each(
  x in array){
    process(x);
  }

Comparing Maps

Serial Map

Parallel Map

- Data
- Task
- Data
- Task
- Data
- Task
- Data
- Task
- Data
- Task
- Data
- Task
- Data
- Task
- Data
Comparing Maps

Serial Map

Parallel Map

Speedup
The space here is speedup. With the parallel map, our program finished execution early, while the serial map is still running.
Independence

- The key to (embarrassing) parallelism is independence

  Warning: No shared state!

  Map function should be “pure” (or “pure-ish”) and should not modify shared states

- Modifying shared state breaks perfect independence

- Results of accidentally violating independence:
  - non-determinism
  - data-races
  - undefined behavior
  - segfaults
Implementation and API

- OpenMP and CilkPlus contain a parallel `for` language construct
- Map is a mode of use of parallel `for`
- TBB uses **higher order functions** with lambda expressions/“funtors”
- Some languages (CilkPlus, Matlab, Fortran) provide **array notation** which makes some maps more concise

Array Notation

```
A[::] = A[::]*5;
```

is CilkPlus array notation for “multiply every element in A by 5”
Often several map operations occur in sequence

- Vector math consists of many small operations such as additions and multiplications applied as maps.

A naïve implementation may write each intermediate result to memory, wasting memory BW and likely overwhelming the cache.

If we fuse the operations used in a sequence of maps into a sequence inside a single map, we can load only the input data at the start of the map and keep intermediate results in registers rather than wasting memory bandwidth on them. We will call this approach code fusion, and it can be applied to other patterns as well. Code fusion is demonstrated in Figure 4.2.
Optimization – Code Fusion

- Can sometimes “fuse” together the operations to perform them at once
- Adds arithmetic intensity, reduces memory/cache usage
- Ideally, operations can be performed using registers alone
Optimization – Cache Fusion

- Sometimes impractical to fuse together the map operations
- Can instead break the work into blocks, giving each CPU one block at a time
- Hopefully, operations use cache alone
Lecture 6
Collectives Pattern
Collectives

- **Collective** operations deal with a *collection* of data as a whole, rather than as separate elements.

- Collective patterns include:
  - Reduce
  - Scan
  - Partition
  - Scatter
  - Gather
Reduce

- **Reduce** is used to combine a collection of elements into one summary value.
- A combiner function combines elements pairwise.
- A combiner function only needs to be *associative* to be parallelizable.
- Example combiner functions:
  - Addition
  - Multiplication
  - Maximum / Minimum
Reduce

Serial Reduction

Parallel Reduction
Reduce

- Vectorization
Reduce

- Tiling is used to break chunks of work up for workers to reduce serially
Reduce

- We can “fuse” the map and reduce patterns
Reduce Example: Dot Product

- 2 vectors of same length
- Map (*) to multiply the components
- Then reduce with (+) to get the final answer

\[ \mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i. \]

Also:

\[ \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \cos(\theta) |\mathbf{b}| \]
Scan

- The **scan** pattern produces partial reductions of input sequence, generates new sequence
- Trickier to parallelize than reduce
- Inclusive scan vs. exclusive scan
  - Inclusive scan: includes current element in partial reduction
  - Exclusive scan: excludes current element in partial reduction, partial reduction is of all prior elements prior to current element
Scan – Example Uses

- Lexical comparison of strings – e.g., determine that “strategy” should appear before “stratification” in a dictionary
- Add multi-precision numbers (those that cannot be represented in a single machine word)
- Evaluate polynomials
- Implement radix sort or quicksort
- Delete marked elements in an array
- Dynamically allocate processors
- Lexical analysis – parsing programs into tokens
- Searching for regular expressions
- Labeling components in 2-D images
- Some tree algorithms – e.g., finding the depth of every vertex in a tree
Scan

- One algorithm for parallelizing scan is to perform an “up sweep” and a “down sweep”
- Reduce the input on the up sweep
- The down sweep produces the intermediate results
Scan

- Just like reduce, we can also fuse the **map** pattern with the **scan** pattern
Lecture 7

Data Reorganization Pattern
Data Movement

- Performance is often more limited by data movement than by computation
  - Transferring data across memory layers is costly
    - locality is important to minimize data access times
    - data organization and layout can impact this
  - Transferring data across networks can take many cycles
    - attempting to minimize the # messages and overhead is important
  - Data movement also costs more in power

- For “data intensive” application, it is a good idea to design the data movement first
  - Design the computation around the data movements
  - Applications such as search and sorting are all about data movement and reorganization
Parallel Data Reorganization

- Remember we are looking to do things in parallel
- How to be faster than the sequential algorithm?
- Similar consistency issues arise as when dealing with computation parallelism
- Here we are concerned more with parallel data movement and management issues
- Might involve the creation of additional data structures (e.g., for holding intermediate data)
Gather Pattern

- Gather pattern creates a (source) collection of data by reading from another (input) data collection
  - Given a collection of (ordered) indices
  - Read data from the source collection at each index
  - Write data to the output collection in index order

- Transfers from source collection to output collection
  - Element type of output collection is the same as the source
  - Shape of the output collection is that of the index collection
    - same dimensionality

- Can be considered a combination of map and random serial read operations
  - Essentially does a number of random reads in parallel
Gather: Defined (parallel perspective)

- Results from the combination of a map with a random read

- Simple pattern, but with many special cases that make the implementation more efficient
Given a collection of read locations
- address or array indices
Gather: Defined

Given a collection of read locations
- address or array indices

and a source array
Gather: Defined

Given a collection of read locations

- address or array indices
and a source array
gather all the data from the source array at the given locations and places them into an **output collection**
Special Case of Gather: Shifts

- Moves data to the left or right in memory
- Data accesses are offset by fixed distances
More about Shifts

- Regular data movement
- Variants from how boundary conditions handled
  - Requires “out of bounds” data at edge of the array
  - Options: default value, duplicate, rotate
- Shifts can be handled efficiently with vector instructions because of regularity
  - Shift multiple data elements at the same time
- Shifts can also take advantage of good data locality
**Special Case of Gather: Zip**

- **Function is to interleaves data (like a zipper)**

Where is the parallelism?

- **FIGURE 6.1** Gather pattern. A collection of data is read from an input collection given a collection of indices. This is equivalent to a map combined with a random read in the map's elemental function.

- **FIGURE 6.2** Shifts are special cases of gather. There are variants based on how boundary conditions are treated. Boundaries can be duplicated, rotated, reflected, a default value can be used, or most generally some arbitrary function can be used. Unlike a general gather, however, shifts can be efficiently implemented using vector instructions since in the interior, the data access pattern is regular.

- **FIGURE 6.3** Zip and unzip (special cases of gather). These operations can be used to convert between array of structures (AoS) and structure of arrays (SoA) data layouts.
Special Case of Gather: Unzip

- Reverses a zip
- Extracts sub-arrays at certain offsets and strides from an input array


gather vs. scatter

gather

- combination of map with random reads
- read locations provided as input

scatter

- combination of map with random writes
- write locations provided as input
- race conditions … why?
**Scatter: Defined**

Given a collection of **input data**
Scatter: Defined

Given a collection of input data and a collection of **write locations**
**Scatter: Defined**

Given a collection of input data and a collection of write locations scatter data to the **output collection**

Problems?
Does the output collection have to be larger in size?
**Scatter: Race Conditions**

Given a collection of input data and a collection of write locations, scatter data to the output collection.

Race Condition: Two (or more) values being written to the same location in output collection. Result is undefined unless enforce rules. **Need rules to resolve collisions!**
Collision Resolution: Atomic Scatter

- **Non-deterministic** approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
Collision Resolution: Merge Scatter

- Collision Resolution: Merge Scatter

- Collision!

- Associative and commutative operators are provided to merge elements in case of a collision.
Collision Resolution: Merge Scatter

- Associative and commutative operators are provided to merge elements in case of a collision.
- Use addition as the merge operator.
- Both associative and commutative properties are required since scatters to a particular location could occur in any order.
Collision Resolution: Priority Scatter

- Every element in the input array is assigned a priority based on its position.
- Priority is used to decide which element is written in case of a collision.
- Example
  - 3D graphics rendering
Converting Scatter to Gather

- Scatter is a more expensive than gather
  - Writing has cache line consequences
  - May cause additional reading due to cache conflicts
  - **False sharing** is a problem that arises
    - writes from different cores go to the same cache line

- Can avoid problems if addresses are know “in advance”
  - Allows optimizations to be applied
  - Convert addresses for a scatter into those for a gather
  - Useful if the same pattern of scatter address will be used repeatedly so the cost is amortized
Pack: Defined

- Used to eliminate unused elements from a collection
- Retained elements are moved so they are contiguous in memory
- Goal is to improve the performance … How?

Diagram:

0 1 1 0 0 1 1 1
A B C D E F G H

X X X

B C F G H

FIGURE 6.9
Pack pattern.
Unpack: Defined

- Inverse of pack operation
- Given the same data on which elements were kept and which were discarded, spread elements back in their original locations

**Figure 6.10** Unpack pattern.

**Figure 6.11** Split pattern.

**Figure 6.12** Unsplit pattern.

Course, split can be emulated with two packs using complementary conditions, but split can usually be implemented more efficiently than two separate packs. Split also does not lose information like pack does. The inverse of split, unsplit, is shown in Figure 6.12. There is some relationship between these patterns and zip and unzip discussed in Section 6.1.3, but unpack are specific patterns that can usually be implemented more efficiently than the more general split and unsplit patterns.
Generalization of Pack: Split

- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

Upper half of output collection: values equal to 0
Generalization of Pack: Split

- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack does.

Lower half of output collection: values equal to 1
Generalization of Pack: Unsplit

- Inverse of split
- Creates **output collection** based on original input collection
Generalization of Pack: Bin

- Generalized split to support more categories (>2)
- Examples
  - Radix sort
  - Pattern classification

4 different categories = 4 bins
Fusion of Map and Pack

- Advantageous if most of the elements of a map are discarded
- Map checks pairs for collision
- Pack stores only actual collisions
- Output BW ~ results reported, not number of pairs tested
- Each element can output 0 or 1 element
Generalization of Pack: Expand

- Each element can output any number of elements
- Results are fused together in order

For example, suppose you wanted to create a parallel implementation of L-system substitution. In L-system substitution, the input and output are strings of characters. Every element of the input string
Parallelizing Algorithms

- Common strategy:
  1. Divide up the computational domain into sections
  2. Work on the sections individually
  3. 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)
Segmentation

- Data is divided into **non-uniform** non-overlapping regions
- Start of each segment can be marked using:
  - Array of integers
  - Array of Boolean flags

![Diagram of segmentation](image)
Lecture 8
Stencil Pattern
Stencil Pattern

- 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.
What is the stencil pattern?
What is the stencil pattern?
What is the stencil pattern?
What is the stencil pattern?

This stencil has 3 elements in the neighborhood: i-1, i, i+1
What is the stencil pattern?

Applies some function to them…

neighborhood

i-1  i  i+1
What is the stencil pattern?

And outputs to the $i^{th}$ position of the output array.
Stencil Patterns

- Stencils can operate on one dimensional and multidimensional data
- Stencil neighborhoods can range from compact to sparse, square to cube, and anything else!
- It is the pattern of the stencil that determines how the stencil operates in an application
2-Dimensional Stencils

4-point stencil
Center cell (P) is not used

5-point stencil
Center cell (P) is used as well

9-point stencil
Center cell (C) is used as well

Source: http://en.wikipedia.org/wiki/Stencil_code
3-Dimensional Stencils

6-point stencil
(7-point stencil)

24-point stencil
(25-point stencil)

Source: http://en.wikipedia.org/wiki/Stencil_code
Stencil Pattern with In Place Update

Input array
Stencil Pattern with In Place Update
Stencil Pattern with In Place Update

Input Array !!!!
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
Consider the following code:

```plaintext
for k=1, 1000
    for i=1, N-2
        for j = 1, N-2
            a[i][j] = 0.25 * (a[i][j] + a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1])
```

Do you see anything interesting?

How would you parallelize?
2-Dimension Jacobi Iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge
- Here we are using a 4-point stencil
- It is different from before because we want to update all array elements simultaneously … How?
2-Dimension Jacobi Iteration

- Consider a 2D array of elements
- Initialize each array element to some value
- At each step, update each array element to the arithmetic mean of its N, S, E, W neighbors
- Iterate until array values converge

![Heat equation simulation](image)

4-point stencil
Successive Over Relaxation (SOR)

- SOR is an alternate method of solving partial differential equations
- While the Jacobi iteration scheme is very simple and parallelizable, its slow convergent rate renders it impractical for any "real world" applications
- One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation
- It also allows a method known as Red-Black SOR to be used to enable parallel updates in place
Red / Black SOR

Pass 1: Writing to red cells, reading from black

Pass 2: Writing to black cells, reading from red
Red / Black SOR
Implementing Stencil with Shift

- One possible implementation of the stencil pattern includes shifting the input data.

- For each offset in the stencil, we gather a new input vector by **shifting** the original input by the offset amount.
Implementing Stencil with Shift

All input arrays are derived from the same original input array.
Implementing Stencil with Shift

- This implementation is only beneficial for one dimensional stencils or the memory-contiguous dimension of a multidimensional stencil.

- Memory traffic to external memory is not reduced with shifts.

- But, shifts allow vectorization of the data reads, which may reduce the total number of instructions.
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.
Conway’s Game of Life

- Working with ghost cells
Conway’s Game of Life

- Working with ghost cells

Five of its eight neighbors already belong to this thread.

But three of its neighbors belong to a different thread.
Conway’s Game of Life

- Working with ghost cells

Before any updates are done in a new iteration, all threads must update their ghost cells.
Conway’s Game of Life

- Working with ghost cells

Data this thread can use (including ghost cells from neighbors)
Conway’s Game of Life

☐ Things to consider…

○ What might happen to our ghost cells as we increase the number of threads?
  ☐ the ghost cells to total cells ratio will rapidly increase causing a greater demand on memory

○ What would be the benefits of using a larger number of ghost cells per thread? Negatives?
  ☐ in the Game of Life example, we could double or triple our ghost cell boundary, allowing us to perform several iterations without stopping for a ghost cell update
Stencil and Communication Optimizations

- When data is distributed, ghost cells must be **explicitly** communicated among nodes between loop iterations.

- Darker cells are PE 0’s ghost cells.

- After first iteration of stencil computation:
  - PE 0 must request PE 1 & PE 2’s stencil results.
  - PE 0 can perform another iteration of 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 lead to increased communication cost
Stencil and Communication Optimizations

- **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 while waiting for ghost cell updates
Recurrence

- This can still be parallelized!
- Trick: find a plane that cuts through grid of intermediate results
  - Previously computed values on one side of plane
  - Values to still be computed on other side of plane
  - Computation proceeds perpendicular to plane through time (this is known as a sweep)
- This plane is called a separating hyperplane
Recurrence

Hyperplanes

Iteration 1
Iteration 2
Iteration 3
Recurrence

- Same grid of intermediate results
- Each level corresponds to a loop iteration
- Computation proceeds downward
Lecture 9
Fork-Join Pattern
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 Concurrency Semantics

- Fork-Join is a concurrency control mechanism
  - Fork increases concurrency
  - Join decreases concurrency

- Fork-Join dependency rules
  - A parent must join with its forked children
  - Forked children with the same parent can join with the parent in any order
  - A child can not join with its parent until it has joined with all of its children

- Fork-Join creates a special type of DAG
  - What do they look like?
Fork Operation

- Fork creates a child thread
- What does the child do?
- Typically, fork operates by assigning the child thread with some piece of “work”
  - Child thread performs the piece of work and then exits by calling join with the parent
- Child work is usually specified by providing the child with a function to call on startup
- Nature of the child work relative to the parent is not specified
Join Operation

- Join informs the parent that the child has finished
- Child thread notifies the parent and then exits
  - Might provide some status back to the parent
- Parent thread waits for the child thread to join
  - Continues after the child thread joins
- Two scenarios
  1. Child joins first, then parent joins with no waiting
  2. Parent joins first and waits, child joins and parent then continues
**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.
Fork-Join Pattern

- Fork-Join directed graph:

  Independent work

  Is it possible for B() and C() to have dependencies between them?
Fork-Join Pattern

- Typical **divide-and-conquer** algorithm implemented with fork-join:

```c
void DivideAndConquer( Problem P ) {
    if( P is base case ) {
        Solve P;
    } else {
        Divide P into K subproblems;
        Fork to conquer each subproblem in parallel;
        Join;
        Combine subsolutions into final solution;
    }
}
```
**Fork-Join Pattern for Divide-Conquer**

- **$K = 2$** (2-way fork-join)
- **$N = 3$** (3 levels of fork-join)
Fork-Join Pattern for Divide-Conquer

$2 \cdot 3 = 8$-way parallelism
Fork-Join Pattern

- Selecting the base case size is critical
- Recursion must go deep enough for plenty of parallelism
- Too deep, and the granularity of sub-problems will be dominated by scheduling overhead
- With K-way fork-join and N levels of fork-join, can have up to $K^N$-way parallelism
Programming Model Support for Fork-Join

- **Cilk Plus:**
  ```cilk
  cilk_spawn B(); — Fork
  C();
  cilk_sync; —— Join
  ```

- B() executes in the child thread
- C() executes in the parent thread
Cilk Plus:

cilk_spawn B();
C();
cilk_sync;

cilk_spawn A();
cilk_spawn B();
cilk_spawn C();
D(); // Not spawned, executed in spawning task

cilk_sync; // Join

for (int i=0; i<n; ++i)
  if (a[i]! =0)
    cilk_spawn f(a[i]);
cilk_sync;
Cilk Plus:

```c
cilk_spawn B();
cilk_spawn C();
/* nil */
cilk_sync;
```

Bad form! Why?
Programming Model Support for Fork-Join

- **TBB**
  - `parallel_invoke()`
    - For 2 to 10 way fork
    - Joins all tasks before returning
  - `Tbb::task_group`
    - For more complicated cases
    - Provides explicit join

```cpp
    task_group g;
    for ( int i=0; i<n; ++i )
        if ( a[i] != 0 )
            g.run( [=,&a]{f(a[i]);} ); // Spawn f(a[i]) as child task
    g.wait(); // Wait for all tasks spawned from g
```
OpenMP:

```c
#pragma omp task
B();
C();
#pragma omp taskwait
```

- Forked task
- Performed by spawning task
OpenMP:

```c
#pragma omp task
B();
C();
#pragma omp taskwait
```

Forked task can also be a compound statement: `{B(); C(); D();}`
Programming Model Support for Fork-Join

- OpenMP:

  ```
  #pragma omp task
  B();
  C();
  #pragma omp taskwait
  ```

  Must be enclosed in an OpenMP `parallel` construct
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
Execution Model of PARALLEL DO

- Master thread executes serial portion of code
- Master thread enters *saxpy* routine
- Master thread encounters *parallel do* directive
- Creates slave threads (How many?)
- Master and slave threads divide iterations of parallel do loop and execute them concurrently
- Implicit synchronization: wait for all threads to finish their allocation of iterations
- Master thread resumes execution after the do loop
- Slave threads disappear

- Abstract execution model – a Fork-Join model!!!
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
```
Assigning Iterations to Threads

- A parallel loop in OpenMP is a worksharing directive
- The manner in which iterations of a parallel loop are assigned to threads is called the loop’s schedule
- Default schedule assigns iterations to threads as evenly as possible (good enough for saxpy)
- Alternative user-specified schedules possible
- More on scheduling later
Parallel versus Parallel Do

- Arbitrary structured blocks versus loops
- Coarse grained versus fine grained
- Replication versus work division (work sharing)

```
!$omp parallel do
  do I = 1,10
    print *, 'Hello world', I
  enddo

!$omp parallel
  do I = 1,10
    print *, 'Hello world', I
  enddo

!$omp end parallel
```

PARALLEL DO is a work sharing directive

Output: 10 Hello world messages

Output: 10*T Hello world messages

where T = number of threads
Work Sharing in Parallel Regions

- Manual division of work (previous example)
- OMP `worksharing` constructs
  - Simplify the programmers job in dividing work among the threads that execute a parallel region
    - `do` directive
      - have different threads perform different iterations of a loop
    - `sections` directive
      - identify sections of work to be assigned to different threads
    - `single` directive
      - specify that a section of code is to be executed by one thread only (remember default is replicated)
DO Directive: Details

- The DO directive does not spawn new threads!
  - It just assigns work to the threads already spawned by the PARALLEL directive

- The work↔thread assignment is identical to that in the PARALLEL DO directive

```c
!$omp parallel do
do I = 1,10
    print *, 'Hello world', I
enddo

!$omp parallel
!$omp do
do I = 1,10
    print *, 'Hello world', I
enddo
!$omp enddo
!$omp end parallel
```
Coarser-Grain Parallelism

- What's going on here? Is this possible? When?
- Is this better? Why?

```c
C$OMP PARALLEL DO
do i=1,n
       .........
enddo
C$OMP PARALLEL DO
do i=1,n
       .........
enddo
C$OMP PARALLEL DO
do i=1,n
       .........
enddo

C$OMP PARALLEL
C$OMP DO
do i=1,n
       .........
enddo
C$OMP DO
do i=1,n
       .........
enddo
C$OMP DO
do i=1,n
       .........
enddo
C$OMP PARALLEL
```
SECTIONS Directive

- Fortran
  ```fortran
  !$omp sections [clause [,] [clause ...]]
  <![[$omp section]
  code for section 1
  <![[$omp section
  code for section 2]

  ...
  <![[$omp end sections [nowait]
  ```

- C/C++
  ```c
  #pragma omp sections [clause [clause ...]]
  {
    [#pragma omp section]
    block

    ...
  }
  ```
**SECTIONS Directive: Details**

- Sections are assigned to threads
  - Each section executes once
  - Each thread executes zero or more sections
- Sections are not guaranteed to execute in any order

```c
#pragma omp parallel
#pragma omp sections
{
    X_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
```
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
Recursive Implementation of Map

cilk_for( unsigned i=lower; i<upper; ++i )
    f(i);

cilk_for can be implemented with a divide-and-conquer routine…

if( lower<upper )
    recursive_map(lower,upper,grainsize,f)
Recursive Implementation of Map

```cpp
template<typename Func>
void recursive_map( unsigned lower, unsigned upper, unsigned grainsize, Func f ) {
    if( upper - lower <= grainsize )
        // Parallel base case
        for( unsigned i = lower; i < upper; ++i )
            f(i);
    else {
        // Divide and conquer
        unsigned middle = lower + (upper - lower) / 2u;
        cilk_spawn recursive_map( lower, middle, grainsize, f );
        recursive_map( middle, upper, grainsize, f );
    }
    // Implicit cilk_sync when function returns
}
```
Recursive Implementation of Map

\[ \text{recursive}_\text{map}(0, 9, 2, f) \]
Choosing Base Cases

- For parallel divide-and-conquer, two base cases:
  - Stopping parallel recursion
  - Stopping serial recursion

- For a machine with $P$ hardware threads, we might think to have $P$ leaves in the spawned functions tree

- This often leads to poor performance
  - Scheduler has no flexibility to balance load
Choosing Base Cases

- Given leaves from spawned function tree with equal work, and equivalent processors, system effects can effect load balance:
  - Page faults
  - Cache misses
  - Interrupts
  - I/O

- Best to over-decompose a problem
- This creates parallel slack
Choosing Base Cases

- **Over-decompose**: parallel programming style where more tasks are specified than there are physical workers. Beneficial in load balancing.

- **Parallel slack**: Amount of extra parallelism available above the minimum necessary to use the parallel hardware resources.
Load Balancing

- Sometimes, threads will finish their work at different rates
- When this happens, some threads may have nothing to do while others may have a lot of work to do
- This is known as a **load balancing** issue
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
Load Balancing

- TBB and Cilk Plus work-stealing differences:

Cilk Plus

TBB

Steal continuation

Steal child
Performance of Fork/Join

Let $A \parallel B$ be interpreted as “fork $A$, do $B$, and join”

Work: $T(A \parallel B)_1 = T(A)_1 + T(B)_1$

Span: $T(A \parallel B)_\infty = \max(T(A)_\infty, T(B)_\infty)$
Cache Locality / Cache-Oblivious Algorithms

- Work/Span analysis ignores memory bandwidth constraints that often limit speedup.
- Cache reuse is important when memory bandwidth is critical resource.
- Tailoring algorithms to optimize cache reuse is difficult to achieve across machines.
- Cache-oblivious programming is a solution for this.
- Code is written to work well regardless of cache structure.
Cache Locality / Cache-Oblivious Algorithms

- Cache-oblivious programming strategy:
  - Recursive divide-and-conquer – good data locality at multiple scales
  - When a problem is subdivided enough, it can fit into the largest cache level
  - Continue subdividing to fit data into smaller and faster cache

- Example problem: matrix multiplication
  - Typical, non-recursive, algorithm uses three nested loops
  - Large matrices won’t fit in cache with this approach
Lecture 10
Pipeline Pattern
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
Pipeline Model

- Stream of data operated on by succession of tasks
- Each task is done in a separate stage

Consider 3 data units and 4 tasks (stages)
- Sequential pipeline execution (no parallel execution)
Where is the Concurrency? (Serial Pipeline)

- Pipeline with serial stages
  - Each stage runs serially (i.e., can not be parallel)
  - Can not parallelize the tasks
- What can we run in parallel?
  - Think about data parallelism
  - Provide a separate pipeline for each data item

- What do you notice as we increase # data items?
Parallelizing Serial Pipelines

- # tasks limits the parallelism with serial tasks

- Two parallel execution choices:
  1) processor executes the entire pipeline
  2) processor assigned to execute a single task

- Which is better? Why?
Pipeline Performance

- N data and T tasks
- Each task takes unit time t
- Sequential time = N*T*t
- Parallel pipeline time = start + finish + (N-2T)/T * t
  = O(N/T) (for N>>T)
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines)
  - Break a larger task into more (shorter) tasks to do
- Interested in pipeline throughput
Pipeline Performance

- \( N \) data and \( T \) tasks
- Suppose the tasks execution times are non-uniform
- Suppose a processor is assigned to execute a task
- What happens to the throughput?
- What limits performance?
- Slowest stage limits throughput … Why?
Pipeline Model with Parallel Stages

- What if we can parallelize a task?
- What is the benefit of making a task run faster?
- Book describes 3 kinds of stages (Intel TBB):
  - Parallel: processing incoming items in parallel
  - Serial out of order: process items 1 at a time (arbitrary)
  - Serial in order: process items 1 at a time (in order)
Serial-Parallel-Serial Pipeline Pattern

• Simplest common sequence of stages for a parallel pipeline
• Serial stages are in order
• Feedback in serial stage indicates that data items are processes in order
• Lack of feedback in parallel stage means data items can be processed in parallel
**Parallel Pipeline Pattern**

- A *pipeline* is composed of several computations called *stages*
  - Parallel stages run independently for each item
  - Serial stages must wait for each item in turn

![Diagram of pipeline](image)

**Figure 9.2**

DAG model of pipeline in Figure 9.1. This picture shows the DAG model of computation, assuming there are five input items. To emphasize the opportunity for parallelism, each box for a parallel task is scaled to show it taking four times as much time as a serial task.
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
Combining Maps

- Map operations are often performed in sequence
- Can sometimes optimize this as one big map
- Not always feasible
  - Steps may be in different modules or libraries
  - There may be serial operations interleaved
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.
Pipelines in TBB

- Built-in support from the `parallel_pipeline` function and the `filter_t` class template
- A `filter_t<X, Y>` takes in type `X` and produces `Y`
  - May be either a serial stage or a parallel stage
- A `filter_t<X, Y>` and a `filter_t<Y, Z>` combine to form a `filter_t<X, Z>`
- `parallel_pipeline()` executes a `filter_t<void, void>`
Pipelines in Cilk Plus

- No built-in support for pipelines
- Implementing by hand can be tricky
  - Can easily fork to move from a serial stage to a parallel stage
  - But can’t simply join to go from parallel back to serial, since workers must proceed in the correct order
  - Could gather results from parallel stage in one big list, but this reduces parallelism and may take too much space
Pipelines in Cilk Plus

- Idea: A reducer can store sub-lists of the results, combining adjacent ones when possible
  - By itself, this would only implement the one-big-list concept
  - However, whichever sub-list is farthest left can process items immediately
    - the list may not even be stored as such; can “add” items to it simply by processing them
  - This way, the serial stage is running as much as possible
  - Eventually, the leftmost sub-list comprises all items, and thus they are all processed
Pipelines in Cilk Plus: Monoids

- Each view in the reducer has a sub-list and an `is_leftmost` flag
- The views are then elements of two monoids*
  - The usual list-concatenation monoid (a.k.a. the *free monoid*), storing the items
  - A monoid* over Booleans that maps $x \otimes y$ to $x$, keeping track of which sub-list is leftmost
    - *Not quite actually a monoid, since a monoid has to have an identity element $I$ for which $I \otimes y$ is always $y$
    - But close enough for our purposes, since the only case that would break is `false` $\otimes$ `true`, and the leftmost view can’t be on the right!

- Combining two views then means concatenating adjacent sub-lists and taking the left `is_leftmost`
Mandatory vs. Optional Parallelism

- Consider a 2-stage (producer-consumer) pipeline
  - Produces puts items into a buffer for consumer
- There is no problem if the producer and consumer run in parallel
- The serial execution is tricky because buffer can fill up and block progress of the producer
  - Similar situation with the consumer
  - Producer and consumer must interleave to guarantee progress
- Restricting the kinds of pipelines that can be built
  - No explicit waiting because a stage invoked only when its input item is ready and must emit exactly 1 output
  - Going from parallel to serial require buffering
- Mandatory parallelism forces the system to execute operations in parallel whereas optional does not require it
The Message-Passing Model

- A process is a program counter and address space
- Processes can have multiple threads (program counters and associated stacks) sharing a single address space

- MPI is for communication among processes (not threads)
- Interprocess communication consists of
  - Synchronization
  - Data movement
SPMD

• Data distributed across processes
  • Not shared

“Owner compute” rule: Process that “owns” the data (local data) performs computations on that data.
Message Passing Programming

- Defined by communication requirements
  - Data communication (necessary for algorithm)
  - Control communication (necessary for dependencies)
- Program behavior determined by communication patterns
- Message passing infrastructure attempts to support the forms of communication most often used or desired
  - Basic forms provide functional access
    - Can be used most often
  - Complex forms provide higher-level abstractions
    - Serve as basis for extension
    - Example: graph libraries, meshing libraries, …
  - Extensions for greater programming power
What is MPI (Message Passing Interface)?

- Message-passing library (interface) specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- Targeted for parallel computers, clusters, and NOWs
  - NOWs = network of workstations
- Specified in C, C++, Fortran 77, F90
- Full-featured and robust
- Designed to access advanced parallel hardware
- End users, library writers, tool developers
- Message Passing Interface (MPI) Forum
  - [http://www.mpi-forum.org/docs/docs.html](http://www.mpi-forum.org/docs/docs.html)
Why Use MPI?

- Message passing is a mature parallel programming model
  - Well understood
  - Efficient match to hardware (interconnection networks)
  - Many applications
- MPI provides a powerful, efficient, and portable way to express parallel programs
- MPI was explicitly designed to enable libraries …
- … which may eliminate the need for many users to learn (much of) MPI
- Need standard, rich, and robust implementation
- Three versions: MPI-1, MPI-2, MPI-3 (just released!)
  - Robust implementations including free MPICH (ANL)
Features of MPI

- General
  - Communicators combine context and group for security
  - Thread safety (implementation dependent)

- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal, synchronous, ready, buffered

- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology
Features of MPI (continued)

- Application-oriented process topologies
  - Built-in support for grids and graphs (based on groups)

- Profiling
  - Hooks allow users to intercept MPI calls
  - Interposition library interface (PMPI)
  - Many tools (e.g., TAU) use PMPI

- Environmental
  - Inquiry
  - Error control
Is MPI Large or Small?

- MPI is large
  - MPI-1 is 128 functions, MPI-2 is 152 functions
  - Extensive functionality requires many functions
  - Not necessarily a measure of complexity

- MPI is small (6 functions)
  - Many parallel programs use just 6 basic functions

- “MPI is just right,” said Baby Bear
  - One can access flexibility when it is required
  - One need not master all parts of MPI to use it
To Use or Not Use MPI?

- **USE**
  - You need a portable parallel program
  - You are writing a parallel library
  - You have irregular or dynamic data relationships that do not fit a data parallel model
  - You care about performance and have to do Exercise 1

- **NOT USE**
  - You don’t need parallelism at all (Ha!)
  - You can use libraries (which may be written in MPI)
  - You can use multi-threading in a concurrent environment
Programming MPI with Only Six Functions

- Many parallel programs can be written using:
  - `MPI_INIT()`
  - `MPI_FINALIZE()`
  - `MPI_COMM_SIZE()`
  - `MPI_COMM_RANK()`
  - `MPI_SEND()`
  - `MPI_RECV()`

- What might be not so great with this?
- Point-to-point (send/recv) isn’t the only way...
  - Add more support for communication
Introduction to Collective Operations in MPI

- Called by all processes in a communicator

- **MPI_BCAST**
  - Distributes data from one process (the root) to all others

- **MPI_REDUCE**
  - Combines data from all processes in communicator
  - Returns it to one process

- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency
Lecture 12
Parallel Algorithms and Parallel Program Design
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
Data Parallel

- Data is decomposed (mapped) onto processors
- Processors performance similar (identical) tasks on data
- Tasks are applied concurrently
- Load balance is obtained through data partitioning
  - Equal amounts of work assigned
- Certainly may have interactions between processors
- Data parallelism scalability
  - Degree of parallelism tends to increase with problem size
  - Makes data parallel algorithms more efficient
- Single Program Multiple Data (SPMD)
  - Convenient way to implement data parallel computation
  - More associated with distributed memory parallel execution
Granularity of Task and Data Decompositions

Granularity can be with respect to tasks and data

Task granularity

- Equivalent to choosing the number of tasks
- Fine-grained decomposition results in large # tasks
- Large-grained decomposition has smaller # tasks
- Translates to data granularity after # tasks chosen
  - consider matrix multiplication

Data granularity

- Think of in terms of amount of data needed in operation
- Relative to data as a whole
- Decomposition decisions based on input, output, input-output, or intermediate data
Tasks Graphs

- 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

\[ \text{Min time} = 27 \quad \text{Min time} = 34 \]

- *Critical path length* bounds parallel execution time
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 = ?
Task Graphs in Action

- Uintah task graph scheduler
  - C-SAFE: Center for Simulation of Accidental Fires and Explosions, University of Utah
  - Large granularity tasks

- PLASMA
  - DAG-based parallel linear algebra
  - DAguge: A generic distributed DAG engine for HPC

DAG of QR for a $4 \times 4$ tiles matrix on a $2 \times 2$ grid of processors.
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?
A Simple Big-Data Problem

- Consider a large data collection of text documents
- Suppose we want to find how often a particular word occurs and determine a probability distribution for all word occurrences

**Sequential algorithm**

<table>
<thead>
<tr>
<th>Data collection</th>
<th>Find and count words</th>
<th>Count words and update statistics</th>
<th>Generate probability distributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>web</td>
<td>2</td>
<td>weed</td>
<td>1</td>
</tr>
<tr>
<td>weed</td>
<td>1</td>
<td>green</td>
<td>2</td>
</tr>
<tr>
<td>green</td>
<td>2</td>
<td>sun</td>
<td>1</td>
</tr>
<tr>
<td>sun</td>
<td>1</td>
<td>moon</td>
<td>1</td>
</tr>
<tr>
<td>moon</td>
<td>1</td>
<td>land</td>
<td>1</td>
</tr>
<tr>
<td>land</td>
<td>1</td>
<td>part</td>
<td>1</td>
</tr>
<tr>
<td>part</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallelization Approach

- **Map**: partition the data collection into subsets of documents and process each subset in parallel
- **Reduce**: assemble the partial frequency tables to derive final probability distribution

**Parallel algorithm**

1. Get next document
2. Find and count words
3. Count words and update statistics
4. Check if more documents
5. Generate probability distributions
Parallelization Approach

- **Map**: partition the data collection into subsets of documents and process each subset in parallel
- **Reduce**: assemble the partial frequency tables to derive final probability distribution

**Parallel algorithm**

- Data collection
- Get next document
- Find and count words
- Count words and update statistics
- Generate probability distributions
- Check if more documents

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>web</td>
<td>2</td>
</tr>
<tr>
<td>weed</td>
<td>1</td>
</tr>
<tr>
<td>green</td>
<td>2</td>
</tr>
<tr>
<td>sun</td>
<td>1</td>
</tr>
<tr>
<td>moon</td>
<td>1</td>
</tr>
<tr>
<td>land</td>
<td>1</td>
</tr>
<tr>
<td>part</td>
<td>1</td>
</tr>
</tbody>
</table>

CIS 410/510: Parallel Computing, University of Oregon, Spring 2014
Map-Reduce Parallel Programming

- Become an important distributed parallel programming paradigm for large-scale applications
  - Also applies to shared-memory parallelism
  - Becomes one of the core technologies powering big IT companies, like Google, IBM, Yahoo and Facebook.
- Framework runs on a cluster of machines and automatically partitions jobs into number of small tasks and processes them in parallel
- Can capture in combining Map and Reduce parallel patterns