Parallel Performance Theory - 2

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

- Scalable parallel execution
- Parallel execution models
- Isoefficiency
- Parallel machine models
- Parallel performance engineering
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
Calculating $\pi$ with Monte Carlo

- Consider a circle of unit radius
- Place circle inside a square box with side of 2 in

- The ratio of the circle area to the square area is:

$$\frac{\pi \times 1 \times 1}{2 \times 2} = \frac{\pi}{4}$$
Monte Carlo Calculation of $\pi$

- Randomly choose a number of points in the square
- For each point $p$, determine if $p$ is inside the circle
- The ratio of points in the circle to points in the square will give an approximation of $\pi/4$
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
  - \( T_1 / (p \cdot T_p) \)
  - \( T_p = \) computation + communication + 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 \text{(\( pT_{par} \) includes overhead)}
  \]
  \[
  T_O(W,p) + W = pT_{par}
  \]
 Isoefficiency Function

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

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

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

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

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

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

- Overhead function:
  - $T_o(W,p) = pT_{par} - W = 2p\log(p)$

- Isoefficiency function:
  - $W = K \times 2p\log(p)$

- 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
  \[ T_0 = p^{3/2} + p^{3/4}W^{3/4} \]

- 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
Parallel Computation (Machine) Models

- **PRAM (parallel RAM)**
  - Basic parallel machine

- **BSP (Bulk Synchronous Parallel)**
  - Isolates regions of computation from communication

- **LogP**
  - Used for studying distribute memory systems
  - Focuses on the interconnection network

- **Roofline**
  - Based in analyzing “feeds” and “speeds”
**PRAM**

- Parallel Random Access Machine (PRAM)
- Shared-memory multiprocessor model
- Unlimited number of processors
  - Unlimited local memory
  - Each processor knows its ID
- Unlimited shared memory
- Inputs/outputs are placed in shared memory
- Memory cells can store an arbitrarily large integer
- Each instruction takes unit time
- Instructions are synchronized across processors (SIMD)
PRAM Complexity Measures

- For each individual processor
  - *Time*: number of instructions executed
  - *Space*: number of memory cells accessed

- PRAM machine
  - *Time*: time taken by the longest running processor
  - *Hardware*: maximum number of active processors

- Technical issues
  - How processors are activated
  - How shared memory is accessed
Processor Activation

- $P_0$ places the number of processors ($p$) in the designated shared-memory cell
  - Each active $P_i$, where $i < p$, starts executing
  - $O(1)$ time to activate
  - All processors halt when $P_0$ halts

- Active processors explicitly activate additional processors via FORK instructions
  - Tree-like activation
  - $O(\log p)$ time to activate
PRAM is a Theoretical (Unfeasible) Model

- Interconnection network between processors and memory would require a very large amount of area.
- The message-routing on the interconnection network would require time proportional to network size.
- Algorithm’s designers can forget the communication problems and focus their attention on the parallel computation only.
- There exist algorithms simulating any PRAM algorithm on bounded degree networks.
- Design general algorithms for the PRAM model and simulate them on a feasible network.
Classification of PRAM Models

- **EREW** (Exclusive Read Exclusive Write)
  - No concurrent read/writes to the same memory location

- **CREW** (Concurrent Read Exclusive Write)
  - Multiple processors may read from the same global memory location in the same instruction step

- **ERCW** (Exclusive Read Concurrent Write)
  - Concurrent writes allowed

- **CRCW** (Concurrent Read Concurrent Write)
  - Concurrent reads and writes allowed

- **CRCW > (ERCW,CREW) > EREW**
CRCW PRAM Models

- COMMON: all processors concurrently writing into the same address must be writing the same value.
- ARBITRARY: if multiple processors concurrently write to the address, one of the competing processors is randomly chosen and its value is written into the register.
- PRIORITY: if multiple processors concurrently write to the address, the processor with the highest priority succeeds in writing its value to the memory location.
- COMBINING: the value stored is some combination of the values written, e.g., sum, min, or max.
- COMMON-CRCW model most often used.
## Complexity of PRAM Algorithms

<table>
<thead>
<tr>
<th>Problem</th>
<th>EREW</th>
<th>CRCW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>List Ranking</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Prefix</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Tree Ranking</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Finding Minimum</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
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 + $g*h + l$
The LogP Model (Culler, Berkeley)

- **Processing**
  - Powerful microprocessor, large DRAM, cache \( \Rightarrow P \)

- **Communication**
  - Significant latency (100's of cycles) \( \Rightarrow L \)
  - Limited bandwidth (1 – 5% of memory) \( \Rightarrow g \)
  - Significant overhead (10's – 100's of cycles) \( \Rightarrow o \)
    - on both ends
    - no consensus on topology
    - should not exploit structure
  - Limited capacity

- No consensus on programming model
  - Should not enforce one
LogP

- **Latency** in sending a (small) message between modules
- **Overhead** felt by the processor on sending or receiving message
- **Gap** between successive sends or receives (1/BW)
- **Processors**

\[
\begin{align*}
\text{L} & \text{ (latency) } \\
\text{o } & \text{ (overhead) } \\
\text{g } & \text{ (gap) }
\end{align*}
\]

Limited Volume (\( L/g \) to or from a processor)
LogP "Philosophy"

- Think about:
  - Mapping of N words onto P processors
  - Computation within a processor
    - its cost and balance
  - Communication between processors
    - its cost and balance
- Characterize processor and network performance
- Do not think about what happens in the network
- This should be enough
## Typical Values for $g$ and $l$

<table>
<thead>
<tr>
<th></th>
<th>$p$</th>
<th>$g$</th>
<th>$l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiprocessor Sun</td>
<td>2-4</td>
<td>3</td>
<td>50-100</td>
</tr>
<tr>
<td>SGI Origin 2000</td>
<td>2-8</td>
<td>10-15</td>
<td>1000-4000</td>
</tr>
<tr>
<td>IBM-SP2</td>
<td>2-8</td>
<td>10</td>
<td>2000-5000</td>
</tr>
<tr>
<td>NOW (Network of Workstations)</td>
<td>2-8</td>
<td>40</td>
<td>5000-20000</td>
</tr>
</tbody>
</table>
Parallel Programming

- To use a scalable parallel computer, you must be able to write parallel programs
- You must understand the programming model and the programming languages, libraries, and systems software used to implement it
- Unfortunately, parallel programming is not easy
Parallel Programming: Are we having fun yet?
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
Parallel Programming: Still a Problem?
Parallel Computing and Scalability

- 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
Parallel Performance and Complexity

- To use a scalable parallel computer well, you must write high-performance parallel programs.
- To get high-performance parallel programs, you must understand and optimize performance for the combination of programming model, algorithm, language, platform, …
- Unfortunately, parallel performance measurement, analysis and optimization can be an easy process.
- Parallel performance is complex.
Parallel Performance Evaluation

- Study of performance in parallel systems
  - Models and behaviors
  - Evaluative techniques

- Evaluation methodologies
  - Analytical modeling and statistical modeling
  - Simulation-based modeling
  - Empirical measurement, analysis, and modeling

- Purposes
  - Planning
  - Diagnosis
  - Tuning
Parallel Performance Engineering and Productivity

- Scalable, optimized applications deliver HPC promise
- Optimization through *performance engineering* process
  - Understand performance complexity and inefficiencies
  - Tune application to run optimally on high-end machines
- How to make the process more effective and productive?
- What performance technology should be used?
  - Performance technology part of larger environment
  - Programmability, reusability, portability, robustness
  - Application development and optimization productivity
- Process, performance technology, and its use will change as parallel systems evolve
- Goal is to deliver effective performance with high productivity value now and in the future
Motivation

- Parallel / distributed systems are complex
  - Four layers
    - application
      - algorithm, data structures
    - parallel programming interface / middleware
      - compiler, parallel libraries, communication, synchronization
    - operating system
      - process and memory management, IO
    - hardware
      - CPU, memory, network
  - Mapping/interaction between different layers
Performance Factors

- Factors which determine a program's performance are complex, interrelated, and sometimes hidden

- Application related factors
  - Algorithms dataset sizes, task granularity, memory usage patterns, load balancing. I/O communication patterns

- Hardware related factors
  - Processor architecture, memory hierarchy, I/O network

- Software related factors
  - Operating system, compiler/preprocessor, communication protocols, libraries
Utilization of Computational Resources

- Resources can be under-utilized or used inefficiently
  - Identifying these circumstances can give clues to where performance problems exist
- Resources may be “virtual”
  - Not actually a physical resource (e.g., thread, process)
- Performance analysis tools are essential to optimizing an application's performance
  - Can assist you in understanding what your program is "really doing"
  - May provide suggestions how program performance should be improved
Performance Analysis and Tuning: The Basics

- Most important goal of performance tuning is to reduce a program's wall clock execution time
  - Iterative process to optimize efficiency
  - Efficiency is a relationship of execution time
- So, where does the time go?
- Find your program's hot spots and eliminate the bottlenecks in them
  - **Hot spot**: an area of code within the program that uses a disproportionately high amount of processor time
  - **Bottleneck**: an area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays
- Understand what, where, and how time is being spent
Sequential Performance

- Sequential performance is all about:
  - How time is distributed
  - What resources are used where and when

- “Sequential” factors
  - Computation
    - choosing the right algorithm is important
    - compilers can help
  - Memory systems and cache and memory
    - more difficult to assess and determine effects
    - modeling can help
  - Input / output
Parallel Performance

- Parallel performance is about sequential performance AND parallel interactions
  - Sequential performance is the performance within each thread of execution
  - “Parallel” factors lead to overheads
    - concurrency (threading, processes)
    - interprocess communication (message passing)
    - synchronization (both explicit and implicit)
  - Parallel interactions also lead to parallelism inefficiency
    - load imbalances
Sequential Performance Tuning

- Sequential performance tuning is a *time-driven* process
- Find the thing that takes the most time and make it take less time (i.e., make it more efficient)
- May lead to program restructuring
  - Changes in data storage and structure
  - Rearrangement of tasks and operations
- May look for opportunities for better resource utilization
  - Cache management is a big one
  - Locality, locality, locality!
  - Virtual memory management may also pay off
- May look for opportunities for better processor usage
Parallel Performance Tuning

- In contrast to sequential performance tuning, parallel performance tuning might be described as conflict-driven or interaction-driven
- Find the points of parallel interactions and determine the overheads associated with them
- Overheads can be the cost of performing the interactions
  - Transfer of data
  - Extra operations to implement coordination
- Overheads also include time spent waiting
  - Lack of work
  - Waiting for dependency to be satisfied
Interesting Performance Phenomena

- Superlinear speedup
  - Speedup in parallel execution is greater than linear
  - $S_p > p$
  - How can this happen?

- Need to keep in mind the relationship of performance and resource usage

- Computation time (i.e., real work) is not simply a linear distribution to parallel threads of execution

- Resource utilization thresholds can lead to performance inflections
Parallel Performance Engineering Process

- Implementation
  - Preparation
    - Performance Analysis
      - Program Tuning
    - Production
  - Measurement
    - Refinement
      - Analysis
    - Ranking