EECS 570
Lecture 2
Message Passing & Shared Memory
Winter 2018
Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Drs. Adve, Falsafi, Martin, Musuvathi, Narayanasamy, Nowatzyk, Wenisch, Sarkar, Mikko Lipasti, Jim Smith, John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Natalie Enright Jerger, Michel Dubois, Murali Annavaram, Per Stenström, and probably others
Announcements

Discussion this Friday.
• Will discuss course project
Readings

For today


For Wednesday 1/13:

Sequential Merge Sort

16MB input (32-bit integers)

Recurse (left)

Recurse (right)

Merge to scratch array

Copy back to input array

Sequential Execution
Parallel Merge Sort (as Parallel Directed Acyclic Graph)

16MB input (32-bit integers)

- Recurse(left)
- Recurse(right)

- Merge to scratch array
- Copy back to input array

Time

Parallel Execution
Parallel DAG for Merge Sort (2-core)

Sequential Sort

Merge

Sequential Sort

Time
Parallel DAG for Merge Sort (4-core)
Parallel DAG for Merge Sort (8-core)
The DAG Execution Model of a Parallel Computation

• Given an input, dynamically create a DAG

• Nodes represent sequential computation
  ▶ Weighted by the amount of work

• Edges represent dependencies:
  ▶ Node A → Node B means that B cannot be scheduled unless A is finished
Sorting 16 elements in four cores
Sorting 16 elements in four cores (4 element arrays sorted in constant time)
Performance Measures

- Given a graph $G$, a scheduler $S$, and $P$ processors

- $T_p(S)$ : Time on $P$ processors using scheduler $S$

- $T_p$ : Time on $P$ processors using best scheduler

- $T_1$ : Time on a single processor (sequential cost)

- $T_\infty$ : Time assuming infinite resources
Work and Depth

- $T_1 = \text{Work}$
  - The total number of operations executed by a computation

- $T_\infty = \text{Depth}$
  - The longest chain of sequential dependencies (critical path) in the parallel DAG
$T_\infty$ (Depth): Critical Path Length (Sequential Bottleneck)
$T_1 \ (work)\ : \ Time \ to \ Run \ Sequentially$
Sorting 16 elements in four cores (4 element arrays sorted in constant time)

Work =
Depth =
Some Useful Theorems
Work Law

• “You cannot avoid work by parallelizing”

\[
\frac{T_1}{P} \leq T_p
\]
**Work Law**

- “You cannot avoid work by parallelizing”

\[ \frac{T_1}{P} \leq T_P \]

Speedup = \[ \frac{T_1}{T_P} \]
Work Law

• “You cannot avoid work by parallelizing”

\[
\frac{T_1}{P} \leq T_P
\]

Speedup = \[
\frac{T_1}{T_P}
\]

• Can speedup be more than 2 when we go from 1-core to 2-core in practice?
Depth Law

• More resources should make things faster
• You are limited by the sequential bottleneck

\[ T_P \geq T_\infty \]
Amount of Parallelism

Parallelism = \frac{T_1}{T_\infty}
Maximum Speedup Possible

Speedup \[ \frac{T_1}{T_P} \leq \frac{T_1}{T_\infty} \] Parallelism

“speedup is bounded above by available parallelism”
Greedy Scheduler

- If more than $P$ nodes can be scheduled, pick any subset of size $P$

- If less than $P$ nodes can be scheduled, schedule them all
Theorem [Graham ’68 & Brent ‘75]. Any greedy scheduler achieves

\[ T_P \leq T_1/P + T_\infty. \]

Proof.

- \# complete steps \leq T_1/P, since each complete step performs \( P \) work.
- \# incomplete steps \leq T_\infty, since each incomplete step reduces the span of the unexecuted dag by 1. ■

Performance of the Greedy Scheduler

\[ T_P(\text{Greedy}) \leq T_1 / P + T_\infty \]

Work law \[ T_1 / P \leq T_P \]

Depth law \[ T_\infty \leq T_P \]
Greedy is optimal within factor of 2

\[ T_P \leq T_P(\text{Greedy}) \leq 2 \ T_P \]

Work law \[ \frac{T_1}{P} \leq T_P \]

Depth law \[ T_{\infty} \leq T_P \]
Work/Depth of Merge Sort (Sequential Merge)

• Work \( T_1 \): \( O(n \log n) \)

• Depth \( T_\infty \): \( O(n) \)
  - Takes \( O(n) \) time to merge \( n \) elements

• Parallelism:
  - \( T_1 / T_\infty = O(\log n) \) → really bad!
Main Message

• Analyze the Work and Depth of your algorithm
• Parallelism is Work/Depth
• Try to decrease Depth
  ▶ the critical path
  ▶ a sequential bottleneck
• If you increase Depth
  ▶ better increase Work by a lot more!
Amdahl’s law

• Sorting takes 70% of the execution time of a sequential program

• You replace the sorting algorithm with one that scales perfectly on multi-core hardware

• How many cores do you need to get a 4x speed-up on the program?
Amdahl’s law, $f = 70\%$

$$\text{Speedup}(f, c) = \frac{1}{1 - f} + \frac{f}{c}$$

- $f$ = the parallel portion of execution
- $1 - f$ = the sequential portion of execution
- $c$ = number of cores used
Amdahl’s law, $f = 70\%$
Amdahl’s law, $f = 70\%$
Amdahl’s law, $f = 10\%$

The graph illustrates the speedup achieved with perfect scaling compared to Amdahl’s law limit, which is just 1.11x.
Amdahl's law, $f = 98\%$
Lesson

- Speedup is limited by **sequential** code

- Even a small percentage of **sequential** code can greatly limit potential speedup
21st Century Computer Architecture

A CCC community white paper
http://cra.org/ccc/docs/init/21stcenturyarchitecturewhitepaper.pdf

Slides from M. Hill, HPCA 2014 Keynote
20th Century ICT Set Up

- Information & Communication Technology (ICT) Has Changed Our World
  - <long list omitted>

- Required innovations in algorithms, applications, programming languages, ..., & system software

- Key (invisible) enablers (cost-)performance gains
  - Semiconductor technology ("Moore’s Law")
  - Computer architecture (~80x per Danowitz et al.)
Enablers: Technology + Architecture

Danowitz et al., CACM 04/2012, Figure 1
21st Century ICT Promises More

Data-centric personalized health care

Computation-driven scientific discovery

Human network analysis

"You never call, and the federal government will back me up on that."

Much more: known & unknown
21st Century App Characteristics

BIG DATA

ALWAYS ONLINE

SECURE/PRIVATE

Whither enablers of future (cost-)performance gains?
Technology’s Challenges 1/2

<table>
<thead>
<tr>
<th>Late 20th Century</th>
<th>The New Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moore’s Law — 2\times transistors/chip</td>
<td>Transistor count still 2\times BUT…</td>
</tr>
<tr>
<td>Dennard Scaling — \sim constant power/chip</td>
<td>Gone. Can’t repeatedly double power/chip</td>
</tr>
</tbody>
</table>
## Technology’s Challenges 2/2

<table>
<thead>
<tr>
<th>Late 20th Century</th>
<th>The New Reality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moore’s Law — 2× transistors/chip</td>
<td>Transistor count still 2× BUT...</td>
</tr>
<tr>
<td>Dennard Scaling — ~constant power/chip</td>
<td>Gone. Can’t repeatedly double power/chip</td>
</tr>
<tr>
<td>Modest (hidden) transistor unreliability</td>
<td>Increasing transistor unreliability can’t be hidden</td>
</tr>
<tr>
<td>Focus on computation over communication</td>
<td>Communication (energy) more expensive than computation</td>
</tr>
<tr>
<td>1-time costs amortized via mass market</td>
<td>One-time cost much worse &amp; want specialized platforms</td>
</tr>
</tbody>
</table>

How should architects step up as technology falters?
“Timeline” from DARPA ISAT

Source: Advancing Computer Systems without Technology Progress, ISAT Outbrief (http://www.cs.wisc.edu/~markhill/papers/isat2012_ACSWTP.pdf)


Approved for Public Release, Distribution Unlimited
The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.
## 21st Century Comp Architecture

<table>
<thead>
<tr>
<th>20th Century</th>
<th>21st Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-chip in generic computer</td>
<td></td>
</tr>
<tr>
<td>Performance via invisible instr.-level parallelism</td>
<td></td>
</tr>
<tr>
<td>Predictable technologies: CMOS, DRAM, &amp; disks</td>
<td></td>
</tr>
</tbody>
</table>

### Cross-Cutting:
- Break current layers with new interfaces
- Performance + security, privacy, availability, programmability, …
- Energy First
  - Parallelism
  - Specialization
  - Cross-layer design

### Predictable technologies:
- CMOS, DRAM, & disks
- New technologies (non-volatile memory, near-threshold, 3D, photonics, …)
- Rethink: memory & storage, reliability, communication
## 21\textsuperscript{st} Century Comp Architecture

<table>
<thead>
<tr>
<th>20\textsuperscript{th} Century</th>
<th>21\textsuperscript{st} Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-chip in generic computer</td>
<td>Architecture as Infrastructure: Spanning sensors to clouds Performance + security, privacy, availability, programmability, ...</td>
</tr>
<tr>
<td>Performance via invisible instr.-level parallelism</td>
<td>Predictable technologies: CMOS, DRAM, &amp; disks</td>
</tr>
</tbody>
</table>

Performance via invisible instr.-level parallelism

Predictable technologies: CMOS, DRAM, & disks
## 21st Century Comp Architecture

<table>
<thead>
<tr>
<th>20th Century</th>
<th>21st Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-chip in generic computer</td>
<td>Architecture Spanning sensors to clouds: Performance, availability, privacy, ...</td>
</tr>
<tr>
<td>Performance via invisible instr.-level parallelism</td>
<td>Energy First:</td>
</tr>
</tbody>
</table>
| Predictable technologies: CMOS, DRAM, & disks | - Parallelism  
- Specialization  
- Cross-layer design |
| Predictable technologies: CMOS, DRAM, & disks | New technologies (non-volatile memory, near-threshold, 3D, photonics, ...) Rethink: memory & storage, reliability, communication |
## 21st Century Comp Architecture

<table>
<thead>
<tr>
<th>20th Century</th>
<th>21st Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-chip in generic computer</td>
<td>Architecture as Infrastructure: Spanning sensors to clouds</td>
</tr>
<tr>
<td>Performance via invisible instr.-level parallelism</td>
<td></td>
</tr>
<tr>
<td>Predictable technologies: CMOS, DRAM, &amp; disks</td>
<td>Performance via invisible instr.-level parallelism</td>
</tr>
</tbody>
</table>

**Predictable technologies:** CMOS, DRAM, & disks

**New technologies:** (non-volatile memory, near-threshold, 3D, photonics, ...) Rethink: memory & storage, reliability, communication
## 21st Century Comp Architecture

<table>
<thead>
<tr>
<th>20th Century</th>
<th>21st Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-chip in stand-alone computer</td>
<td><strong>Architecture as Infrastructure:</strong></td>
</tr>
<tr>
<td></td>
<td>Spanning sensors to clouds</td>
</tr>
<tr>
<td></td>
<td>Performance + security, privacy, availability, programmability, …</td>
</tr>
<tr>
<td>Performance via invisible instr.-level parallelism</td>
<td><strong>Energy First</strong></td>
</tr>
<tr>
<td></td>
<td>• Parallelism</td>
</tr>
<tr>
<td></td>
<td>• Specialization</td>
</tr>
<tr>
<td></td>
<td>• Cross-layer design</td>
</tr>
<tr>
<td>Predictable technologies: CMOS, DRAM, &amp; disks</td>
<td><strong>New technologies</strong> (non-volatile memory, near-threshold, 3D, photonics, …) Rethink: memory &amp; storage, reliability, communication</td>
</tr>
<tr>
<td></td>
<td><strong>Cross-Cutting:</strong></td>
</tr>
<tr>
<td></td>
<td>Break current layers with new interfaces</td>
</tr>
</tbody>
</table>
Cost-Effective Computing
[Wood & Hill, IEEE Computer 1995]

Premise: Isn’t speedup(P) < P inefficient?
- If only throughput matters, use P computers instead…

• Key observation: much of a computer’s cost is NOT CPU
Let Costup(P) = Cost(P)/Cost(1)
Parallel computing is cost-effective if:
- Speedup(P) > Costup(P)

• E.g., for SGI PowerChallenge w/ 500 MB
  - Costup(32) = 8.6
Parallel Programming Models and Interfaces
Programming Models

• High level paradigm for expressing an algorithm
  □ Examples:
    ○ Functional
    ○ Sequential, procedural
    ○ Shared memory
    ○ Message Passing

• Embodied in languages that support concurrent execution
  □ Incorporated into language constructs
  □ Incorporated as libraries added to existing sequential language

• Top level features:
  (For conventional models – shared memory, message passing)
  □ Multiple threads are conceptually visible to programmer
  □ Communication/synchronization are visible to programmer
An Incomplete Taxonomy

- VLIW
- SIMD
- Vector
- Data Flow
- GPU
- MapReduce / WSC
- Systolic Array
- Reconfigurable / FPGA
- ...

- Emerging & Lunatic Fringe
- Shared Memory
  - Bus-based
  - Switching Fabric-based
    - Incoherent
    - Coherent
- Message Passing
  - COTS-based
  - Switching Fabric-based

MP Systems
Programming Model Elements

• For both Shared Memory and Message Passing
• Processes and threads
  - Process: A shared address space and one or more threads of control
  - Thread: A program sequencer and private address space
  - Task: Less formal term – part of an overall job
  - Created, terminated, scheduled, etc.
• Communication
  - Passing of data
• Synchronization
  - Communicating control information
  - To assure reliable, deterministic communication
Historical View

Join at: I/O (Network)
Program with: Message passing

Memory
Shared Memory

Processor
Dataflow, SIMD, VLIW, CUDA, other data parallel
Message Passing Programming Model
Message Passing Programming Model

- User level send/receive abstraction
  - Match via local buffer \((x,y)\), process \((Q,P)\), and tag \((t)\)
  - Need naming/synchronization conventions
Message Passing Architectures

- Cannot directly access memory of another node
- IBM SP-2, Intel Paragon, Myrinet Quadrics QSW
- Cluster of workstations (e.g., MPI on flux cluster)
MPI - Message Passing Interface API

- A widely used standard
  - For a variety of distributed memory systems
    - SMP Clusters, workstation clusters, MPPs, heterogeneous systems
- Also works on Shared Memory MPs
  - Easy to emulate distributed memory on shared memory HW
- Can be used with a number of high level languages
- Available in the Flux cluster at Michigan
Processes and Threads

- Lots of flexibility (advantage of message passing)
  1. Multiple threads sharing an address space
  2. Multiple processes sharing an address space
  3. Multiple processes with different address spaces
     - and different OSes

- 1 and 2 easily implemented on shared memory HW (with single OS)
  - Process and thread creation/management similar to shared memory

- 3 probably more common in practice
  - Process creation often external to execution environment; e.g. shell script
  - Hard for user process on one system to create process on another OS
Communication and Synchronization

- Combined in the message passing paradigm
  - Synchronization of messages part of communication semantics
- Point-to-point communication
  - From one process to another
- Collective communication
  - Involves groups of processes
  - e.g., broadcast
Message Passing: Send()

- Send( <what>, <where-to>, <how> )

  - What:
    - A data structure or object in user space
    - A buffer allocated from special memory
    - A word or signal

  - Where-to:
    - A specific processor
    - A set of specific processors
    - A queue, dispatcher, scheduler

  - How:
    - Asynchronously vs. synchronously
    - Typed
    - In-order vs. out-of-order
    - Prioritized
Message Passing: Receive()

- Receive( <data>, <info>, <what>, <how> )

  - Data: mechanism to return message content
    - A buffer allocated in the user process
    - Memory allocated elsewhere

  - Info: meta-info about the message
    - Sender-ID
    - Type, Size, Priority
    - Flow control information

  - What: receive only certain messages
    - Sender-ID, Type, Priority

  - How:
    - Blocking vs. non-blocking
Synchronous vs Asynchronous

- Synchronous Send
  - Stall until message has actually been received
  - Implies a message acknowledgement from receiver to sender

- Synchronous Receive
  - Stall until message has actually been received

- Asynchronous Send and Receive
  - Sender and receiver can proceed regardless
  - Returns *request handle* that can be tested for message receipt
  - Request handle can be tested to see if message has been sent/received
Deadlock

• Blocking communications may deadlock

  <Process 0>
  Send(Process1, Message);
  Receive(Process1, Message);

  <Process 1>
  Send(Process0, Message);
  Receive(Process0, Message);

• Requires careful (safe) ordering of sends/receives

  <Process 0>
  Send(Process1, Message);
  Receive(Process1, Message);

  <Process 1>
  Receive (Process0, Message);
  Send (Process0, Message);
Message Passing Paradigm Summary

Programming Model (Software) point of view:

• Disjoint, separate name spaces
• “Shared nothing”
• Communication via explicit, typed messages: send & receive
Message Passing Paradigm Summary

Computer Engineering (Hardware) point of view:

• Treat inter-process communication as I/O device

• Critical issues:
  □ How to optimize API overhead
  □ Minimize communication latency
  □ Buffer management: how to deal with early/unsolicited messages, message typing, high-level flow control
  □ Event signaling & synchronization
  □ Library support for common functions (barrier synchronization, task distribution, scatter/gather, data structure maintenance)
Shared Memory Programming Model
Shared-Memory Model

- Multiple execution contexts sharing a single address space
  - Multiple programs (MIMD)
  - Or more frequently: multiple copies of one program (SPMD)
- Implicit (automatic) communication via loads and stores
- Theoretical foundation: PRAM model

Memory System

- $P_1$
- $P_2$
- $P_3$
- $P_4$
Global Shared Physical Address Space

- Communication, sharing, synchronization via loads/stores to shared variables
- Facilities for address translation between local/global address spaces
- Requires OS support to maintain this mapping

Shared portion of address space

Private portion of address space

Common physical address space

- \( P_n \) private
- \( P_2 \) private
- \( P_1 \) private
- \( P_0 \) private

load \( P_n \)

store \( P_0 \)
Why Shared Memory?

Pluses
- For applications looks like multitasking uniprocessor
- For OS only evolutionary extensions required
- Easy to do communication without OS
- Software can worry about correctness first then performance

Minuses
- Proper synchronization is complex
- Communication is implicit so harder to optimize
- Hardware designers must implement

Result
- Traditionally bus-based Symmetric Multiprocessors (SMPs), and now CMPs are the most success parallel machines ever
- And the first with multi-billion-dollar markets
Thread-Level Parallelism

```c
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)
{
    accts[id].bal -= amt;
    spew_cash();
}
```

• **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared loosely, dynamically

• Example: database/web server (each query is a thread)
  - `accts` is shared, can’t register allocate even if it were scalar
  - `id` and `amt` are private variables, register allocated to `r1, r2`
Synchronization

- Mutual exclusion: locks, ...
- Order: barriers, signal-wait, ...

- Implemented using read/write/modify to shared location
  - Language-level:
    - libraries (e.g., locks in pthread)
    - Programmers can write custom synchronizations
  - Hardware ISA
    - E.g., test-and-set

- OS provides support for managing threads
  - scheduling, fork, join, futex signal/wait

We’ll cover synchronization in more detail in a few weeks
Paired vs. Separate Processor/Memory?

• Separate processor/memory
  • Uniform memory access (UMA): equal latency to all memory
    + Simple software, doesn’t matter where you put data
    - Lower peak performance
  • Bus-based UMAs common: symmetric multi-processors (SMP)

• Paired processor/memory
  • Non-uniform memory access (NUMA): faster to local memory
    - More complex software: where you put data matters
    + Higher peak performance: assuming proper data placement
**Shared vs. Point-to-Point Networks**

- **Shared network**: e.g., bus (left)
  - Low latency
  - Low bandwidth: doesn’t scale beyond ~16 processors
  - Shared property simplifies cache coherence protocols (later)

- **Point-to-point network**: e.g., mesh or ring (right)
  - Longer latency: may need multiple “hops” to communicate
  - Higher bandwidth: scales to 1000s of processors
  - Cache coherence protocols are complex
Implementation #1: Snooping Bus MP

- Two basic implementations
- Bus-based systems
  - Typically small: 2–8 (maybe 16) processors
  - Typically processors split from memories (UMA)
    - Sometimes multiple processors on single chip (CMP)
    - Symmetric multiprocessors (SMPs)
    - Common, I use one everyday
Implementation #2: Scalable MP

- General point-to-point network-based systems
  - Typically processor/memory/router blocks (NUMA)
    - **Glueless MP**: no need for additional “glue” chips
  - Can be arbitrarily large: 1000’s of processors
    - **Massively parallel processors (MPPs)**
  - In reality only government (DoD) has MPPs...
    - Companies have much smaller systems: 32–64 processors
    - **Scalable multi-processors**
Cache Coherence

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)
No-Cache, No-Problem

- Scenario I: processors have no caches
  - No problem
Cache Incoherence

- Scenario II: processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (inconsistent)
Snooping Cache-Coherence Protocols

Bus provides serialization point

Each cache controller “snoops” all bus transactions

- take action to ensure coherence
  - invalidate
  - update
  - supply value
- depends on state of the block and the protocol
Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

  - **Part I**: bus bandwidth
    - Replace non-scalable bandwidth substrate (bus)...
    - ...with scalable bandwidth one (point-to-point network, e.g., mesh)

  - **Part II**: processor snooping bandwidth
    - Interesting: most snoops result in no action
    - Replace non-scalable broadcast protocol (spam everyone)...
    - ...with scalable directory protocol (only spam processors that care)

- We will cover this in Unit 3
Shared Memory Summary

• Shared-memory multiprocessors
  + Simple software: easy data sharing, handles both DLP and TLP
    – Complex hardware: must provide illusion of global address space

• Two basic implementations
  ❑ Symmetric (UMA) multi-processors (SMPs)
    ○ Underlying communication network: bus (ordered)
      + Low-latency, simple protocols that rely on global order
        – Low-bandwidth, poor scalability
  ❑ Scalable (NUMA) multi-processors (MPPs)
    ○ Underlying communication network: point-to-point (unordered)
      + Scalable bandwidth
        – Higher-latency, complex protocols