Parallel and Distributed Systems

Instructor: Sandhya Dwarkadas
Department of Computer Science
University of Rochester

What is a parallel computer?
“A collection of processing elements that communicate and cooperate to solve large problems fast”

What is a distributed system?
“A collection of independent computers that appear to its users as a single coherent system”

Why Parallel or Distributed Computing?

• Fundamentally limited by the speed of a sequential processor
• Resource sharing
• Information exchange
• Collaboration

Hardware Trends

Moore’s law (attributed to Gordon Moore, Founder of Intel):
Circuit complexity double every 18 months (revised 1975)

Image from http://www.intel.com/technology/silicon/mooreslaw/pix/mooreslaw_chart.gif
Leveraging Moore’s Law

• More transistors – opportunities for exploiting parallelism
  – Implicit parallelism
    • Pipelining
    • Superscalar
  – Explicit parallelism
    • Streaming and multimedia processor extensions
      – E.g., MMX, Altivec
    • Very long instruction words (VLIW)

Current Trends

• Problems:
  – Fundamental circuit delay and heat limitations
  – Limited amount of instruction-level parallelism
• Solutions: proliferation of (from Sun, IBM, Intel, Nvidia, …)
  – Accelerators
  – Multithreading
  – Multicore
  – Multiprocessors
Qualcomm’s Snapdragon, NVIDIA’s Tegra

Top 500 List of Supercomputers (www.top500.org – Nov. 2015)

- Top 5 from the list
  - Tianhe-2, Intel Xeon and Xeon Phi at NSCC China, 3,120,000 cores, 33.8 (54.9 peak) Petaflops/sec on Linpack, 17.8 MW
  - Titan, Cray XK7 at Oakridge (ORNL), AMD Opteron and NVIDIA, 560,640 cores, 17.6 (27 peak) Pflops/sec, 8.2 MW
  - Sequoia, BlueGene/Q, LLNL, 1,572,864 cores, 17.2 (20 peak) Pflops/sec, 7.9 MW
  - Fujitsu’s K, RIKEN AICS Japan, SPARC, 705,024 cores, TOFU Interconnect, 10.5 Pflops/sec, 12.6 MW
  - Mira, IBM BlueGene/Q at ANL, 786,432 cores, 8.38 Pflops/sec, 3.9 MW

Top 500 List of Supercomputers (www.top500.org – Nov. 2014)

- Top 5 from the list
  - Tianhe-2, Intel Xeon and Xeon Phi, 3,120,000 cores, 33.8 (54.0 peak) Petaflops/sec on Linpack, 17.8 MW
  - Titan, Cray XK7 at Oakridge (ORNL), AMD Opteron and NVIDIA, 560,640 cores, 17.59 (27 peak) Pflops/sec, 8 MW
  - Sequoia, BlueGene/Q, LLNL, 1,572,864 cores, 17.1 Pflops/sec, 8 MW
  - Fujitsu’s K, RIKEN, Japan, SPARC, 705,024 cores, TOFU Interconnect, 10.51 Pflops/sec, 12.6 MW
  - Mira, IBM BlueGene/Q at ANL, 786.432 cores, 8.58 Pflops/sec, 3.9 MW

Intel’s Many-Integrated-Cores (MIC/Phi)

- 60 cores, 240 threads
- 8 GB memory, 320 GB/s bandwidth
- 1 Teraflop peak double-precision floating point

Previously requiring 2,500 sq. ft!
Top 500 List of Supercomputers
(www.top500.org – Nov. 2012)

- Top 5 from the list (23 with petaflops performance)
  - Titan, Cray XK7 at Oakridge (ORNL), 560,640 cores, 17.59 Petaflops/sec on Linpack
  - Sequoia, BlueGene/Q, LLNL, 1,572,864 cores, 16.32 Pflops/sec
  - Fujitsu’s K, RIKEN, Japan, 705,024 cores, 10.51 Pflops/sec
  - Mira, IBM BlueGene/Q at ANL, 786,432 cores, 8.16 Pflops/sec
  - JUQueen, IBM Blue Gene/Q in Germany, 393,216 cores, 4.14 Pflops/sec

Top 500 List of Supercomputers
(www.top500.org – Nov. 2008)

- Top 5 from the list
  - IBM Cell-based “Roadrunner”, 2008, 129600 cores, 1.059 petaflops/sec
  - Cray XT5 “Jaguar”, 2008, 150152 cores
  - SGI “Pleides”, 2008, 51200 cores
  - IBM BlueGene/L, 2007, 212992 cores
  - IBM Blue Gene/P, 2007, 163840 cores

Parallel Systems Issues

- Data sharing – single versus multiple address spaces
- Process coordination – synchronization using locks, messages, …
- Distributed (non-uniform access – NUMA) versus centralized (uniform access – UMA) memory
- Connectivity – single shared bus versus network with many different topologies
- Fault tolerance/reliability

Shared Memory Parallel Systems

- Multiple processors can access the same memory simultaneously
- Challenges:
  - One processor’s cache may contain a copy of data that was just modified by another
    - Requires hardware support for coherence
  - Two processes’ operations may be interleaved in unintuitive ways
    - Requires hardware support and guarantees on atomicity and ordering
Multicore Resource Management

- Multicore systems contain many shared resources, e.g., memory bandwidth and cache
- Problem:
  - Management of shared resources for
    - Efficiency, fairness

Distributed Memory Parallel Systems

- Parallel systems that do not share memory
- Software system support for communication (point-to-point, group)
- Data must be explicitly partitioned and transferred when needed
- Dynamic workload management?

Distributed Systems Issues

- Transparency
  - Access (data representation), location, migration, relocation, replication, concurrency, failure, persistence
- Scalability
  - Size, geographically, administratively
- Reliability/fault tolerance: fail-stop, byzantine (arbitrary) failure models

Distributed Systems Issues

- Problems:
  - Distributed consensus
  - Replication, caching consistency
  - Security and trust
Cluster-Based Servers

- Large Internet data centers (Google, Microsoft, …)
  - A form of parallel computing
  - Customized parallel programming model (MapReduce)
  - A distributed system
    - Consistency, reliability, scalability, availability
  - Power management, network topology, error monitoring

Parallel Programming

- Challenges:
  - Control dependences
  - Data sharing

Parallel Programming Models

- Data parallel – HPF, Fortran-D, Power C/Fortran
- Shared memory - pthreads
- Message passing – MPI, PVM
- Global address space

Parallelism

- Ability to execute different parts of a program concurrently on different machines
- Goal: shorten execution time
- Grain of parallelism: size of parts
- Focus on coarse grain (not instruction), although examples will be fine grain for simplicity
Why is Parallel Computing Hard?

- Amdahl’s law – insufficient available parallelism
  - Speedup = 1/(fraction_enhanced/speedup + (1-fraction_enhanced))
- Overhead of communication and coordination
- Portability – knowledge of underlying architecture often required

Steps in the Parallelization Process

- Decomposition into tasks
- Assignment to processes
- Orchestration – communication of data, synchronization among processes

Basics of Parallelization

- Dependence analysis
- Synchronization
  - Events
  - Mutual exclusion
- Parallelism patterns

When can two statements execute in parallel?

- On one processor:
  - statement 1;
  - statement 2;
- On two processors:
  - processor1:
    - statement 1;
  - processor2:
    - statement 2;
Fundamental Assumption

• Processors execute independently: no control over order of execution between processors

When can 2 statements execute in parallel?

• Possibility 1
  Processor1: statement1;
  Processor2: statement2;

• Possibility 2
  Processor1: statement2;
  Processor2: statement1;

When can 2 statements execute in parallel?

• Their order of execution must not matter!

• In other words, statement1; statement2;
  must be equivalent to statement2; statement1;

Example 1

a = 1;
b = 2;

• Statements can be executed in parallel.
Example 2

```plaintext
a = 1;
b = a;
```

- Statements cannot be executed in parallel
- Program modifications may make it possible.

Example 3

```plaintext
a = f(x);
b = a;
```

- May not be wise to change the program (sequential execution would take longer).

Example 4

```plaintext
b = a;
a = 1;
```

- Statements cannot be executed in parallel.

Example 5

```plaintext
a = 1;
a = 2;
```

- Statements cannot be executed in parallel.
True dependence
Statements S1, S2
S2 has a true dependence on S1
iff
S2 reads a value written by S1

Anti-dependence
Statements S1, S2.
S2 has an anti-dependence on S1
iff
S2 writes a value read by S1.

Output Dependence
Statements S1, S2.
S2 has an output dependence on S1
iff
S2 writes a variable written by S1.

When can 2 statements execute in parallel?
S1 and S2 can execute in parallel
iff
there are no dependences between S1 and S2
– true dependences
– anti-dependences
– output dependences
Some dependences can be removed.
Example 6

• Parallelism often occurs in loops.

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

• No dependences.
• Iterations can be executed in parallel.

Example 7

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

Iterations and statements can be executed in parallel.

Example 8

```c
for(i=0; i<100; i++) a[i] = i;
for(i=0; i<100; i++) b[i] = 2*i;
```

Iterations and loops can be executed in parallel.

Example 9

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

• There is a dependence … on itself!
• Loop is still parallelizable.
Example 10

for( i=0; i<100; i++ )
    a[i] = f(a[i-1]);

• Dependence between a[i] and a[i-1].
• Loop iterations are not parallelizable.

Loop-carried dependence

• A loop carried dependence is a dependence that is present only if the statements are part of the execution of a loop.
• Otherwise, we call it a loop-independent dependence.
• Loop-carried dependences prevent loop iteration parallelization.

Example 11

for(i=0; i<100; i++)
    for(j=0; j<100; j++)
        a[i][j] = f(a[i][j-1]);

• Loop-independent dependence on i.
• Loop-carried dependence on j.
• Outer loop can be parallelized, inner loop cannot.

Example 12

for( j=0; j<100; j++ )
    for( i=0; i<100; i++ )
        a[i][j] = f(a[i][j-1]);

• Inner loop can be parallelized, outer loop cannot.
• Less desirable situation (finer-grain parallelism).
• Loop interchange is sometimes possible.
Level of loop-carried dependence

• Is the nesting depth of the loop that carries the dependence.
• Indicates which loops can be parallelized.

Be careful … Example 13

printf("a");
printf("b");

Statements have a hidden output dependence due to the output stream.

Be careful … Example 14

a = f(x);
b = g(x);

Statements could have a hidden dependence if f and g update the same variable.

Be careful … Example 15

for(i=0; i<100; i++)
a[i+10] = f(a[i]);

• Dependence between a[10], a[20], …
• Dependence between a[11], a[21], …
• …
• Some parallel execution is possible.
Be careful … Example 16

```c
for( i=1; i<100;i++ ) {
    a[i] = ...;
    ... = a[i-1];
}
```

- Dependence between a[i] and a[i-1]
- Complete parallel execution impossible
- Pipelined parallel execution possible

Be careful … Example 17

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

- Cannot tell for sure.
- Parallelization depends on user knowledge of values in indexa[].
- User can tell, compiler cannot.

An aside

- Parallelizing compilers analyze program dependences to decide parallelization.
- In parallelization by hand, user does the same analysis.
- Compiler more convenient and more correct
- User more powerful, can analyze more patterns.

To remember

- Statement order must not matter.
- Statements must not have dependences.
- Some dependences can be removed.
- Some dependences may not be obvious.
Types of Dependences

- Flow (or True) dependence – RAW
- Anti-dependence – WAR
- Output dependence – WAW

When can 2 statements execute in parallel?

S1 and S2 can execute in parallel iff there are no dependences between S1 and S2
- true dependences
- anti-dependences
- output dependences

Some dependences can be removed.

Loop-Carried Dependence

- A loop-carried dependence is a dependence that is present only if the statements occur in two different instances of a loop
- Otherwise, we call it a loop-independent dependence
- Loop-carried dependences limit loop iteration parallelization

Synchronization

- Used to enforce dependences
- Control the ordering of events on different processors
  - Events – signal(x) and wait(x)
  - Fork-Join or barrier synchronization (global)
  - Mutual exclusion/critical sections
Example 1: Creating Parallelism by Enforcing Dependences

```c
for( i=1; i<100; i++ ) {
    a[i] = ...;
    ...;
    ... = a[i-1];
}
```

• Loop-carried dependence, not parallelizable

Synchronization Facility

• Suppose we had a set of primitives, `signal(x)` and `wait(x)`.
• `wait(x)` blocks unless a `signal(x)` has occurred.
• `signal(x)` does not block, but causes a `wait(x)` to unblock, or causes a future `wait(x)` not to block.

Example 1: Enforcing Dependencies (continued)

```c
for( i=...; i<...; i++ ) {
    a[i] = ...;
    signal(e_a[i]);
    ...;
    wait(e_a[i-1]);
    ... = a[i-1];
}
```

Example 1 (continued)

• Note that here it matters which iterations are assigned to which processor.
• It does not matter for correctness, but it matters for performance.
• Cyclic assignment is probably best.
Course Overview

• Parallel programming
  – models, languages, and algorithms

• Parallel systems design issues
  – E.g., synchronization, communication, coherence, consistency

• Distributed systems design issues
  – E.g., consistency, group communication, fault tolerance