Chapter 5: Multiprocessors (Thread-Level Parallelism) – Part 1

Introduction

What is a parallel or multiprocessor system?

Why parallel architecture?

Performance potential

Flynn classification

Communication models

Architectures

Centralized shared-memory

Distributed shared-memory

Parallel programming

Synchronization

Memory consistency models
What is a parallel or multiprocessor system?

Multiple processor units working together to solve the same problem

Key architectural issue: Communication model
Why parallel architectures?

*Absolute performance*
  - Scientific computing
  - General-purpose computing

Technology and architecture trends in high-performance computing
  - # of transistors on chip growing rapidly
  - Clock rates, ILP leveling out
    - Complexity, power!

⇒ Multicore chips from almost all vendors!
  - Connect multicore together for even more parallelism
Amdahl's Law is pessimistic

Let $s$ be the serial part

Let $p$ be the part that can be parallelized $n$ ways

Serial: $\text{SSPPPPPP}$

6 processors: $\text{SSP}$

\[ \text{P} \]

\[ \text{P} \]

\[ \text{P} \]

\[ \text{P} \]

\[ \text{P} \]

\[ \text{P} \]

Speedup $= \frac{8}{3} = 2.67$

\[ T(n) = \frac{1}{s+p/n} \]

As $n \to \infty$, $T(n) \to \frac{1}{s}$

Pessimistic
Gustafson's Corollary

Amdahl's law holds if run same problem size on larger machines
But, in practice, people run larger problems and "wait" the same time
Assume for larger problem sizes
Serial time fixed (at s)
Parallel time proportional to problem size (truth more complicated)

Old Serial: SSPPPPPP
6 processors: SSPPPPPP
             PPPPPP
             PPPPPP
             PPPPPP
             PPPPPP
             PPPPPP
Hypothetical Serial:
  SSPPPPPP PPPPPP PPPPPP PPPPPP PPPPPP PPPPPP

\[
\text{Speedup} = (8+5\times6)/8 = 4.75
\]
\[
T'(n) = s + n*p; T'(\infty) \to \infty!!!!
\]
How does your algorithm "scale up"?
Flynn classification

- Single-Instruction Single-Data (SISD)
- Single-Instruction Multiple-Data (SIMD)
- Multiple-Instruction Single-Data (MISD)
- Multiple-Instruction Multiple-Data (MIMD)
Communication models

Shared-memory
Message passing
Data parallel
Communication Models: Shared-Memory

Each node a processor that runs a process

One shared memory
  Accessible by any processor
  The same address on two different processors refers to the same datum

Therefore, write and read memory to
  Store and recall data
  Communicate, Synchronize (coordinate)
Communication Models: Message Passing

Each node a computer
  Processor – runs its own program (like SM)
  Memory – local to that node, unrelated to other memory
Add messages for internode communication, send and receive like mail
Virtual processor per datum

Write sequential programs with "conceptual PC" and let parallelism be within the data (e.g., matrices)

\[ C = A + B \]

Typically SIMD architecture, but MIMD can be as effective
Architectures

All mechanisms can usually be synthesized by all hardware

Key: which communication model does hardware support best?

All small-scale systems are shared-memory
**Which is Best Communication Model to Support?**

**Shared-memory**
- Used in small-scale systems
- Easier to program for dynamic data structures
- Lower overhead communication for small data
- Implicit movement of data with caching
- Hard to build?

**Message-passing**
- Communication explicit longer to program?
- Larger overheads in communication OS intervention?
- Easier to build?
For now, assume interconnect is a bus – *centralized architecture*
Centralized Shared-Memory Architecture

PROC

PROC

PROC

MEMORY

BUS

...
Centralized Shared-Memory Architecture (Cont.)

For higher bandwidth (throughput)

For lower latency

Problem?
**Centralized Shared-Memory Architecture (Cont.)**

For higher bandwidth (throughput)

![Diagram showing multiple processors and memories connected to a bus](image)

For lower latency

Problem?
Centralized Shared-Memory Architecture (Cont.)**

For higher bandwidth (throughput)

For lower latency
Cache Coherence Problem

Diagram showing the connection between PROC 1, PROC 2, PROC n, CACHE, MEMORY, and the BUS.
Cache Coherence Solutions

Snooping

PROC 1 → CACHE → BUS → MEMORY

PROC 2 → CACHE → BUS → MEMORY

PROC n → CACHE → BUS → MEMORY

Problem with centralized architecture
Distributed Shared-Memory (DSM) Architecture

Use a higher bandwidth interconnection network

PROC 1

CACHE

PROC 2

CACHE

PROC n

CACHE

GENERAL INTERCONNECT

MEMORY

MEMORY

MEMORY

Uniform memory access architecture (UMA)
Distributed Shared-Memory (DSM) - Cont.

For lower latency: Non-Uniform Memory Access architecture (NUMA)
Distributed Shared-Memory (DSM) -- Cont.**

For lower latency: Non-Uniform Memory Access architecture (NUMA)
Non-Bus Interconnection Networks

Example interconnection networks
Distributed Shared-Memory - Coherence Problem

Directory scheme

PROC -> MEM -> CACHE

PROC -> MEM -> CACHE

PROC -> MEM -> CACHE

... ... ...

PROC -> MEM -> CACHE

SWITCH/NETWORK

Level of indirection!
Directory scheme

PROC -> MEM -> CACHE

PROC -> MEM -> CACHE

PROC -> MEM -> CACHE

PROC -> MEM -> CACHE

SWITCH/NETWORK

Level of indirection!
Parallel Programming Example

Add two matrices: \( C = A + B \)

Sequential Program

```c
main(argc, argv)
int argc; char *argv;
{
    Read(A);
    Read(B);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            C[i, j] = A[i, j] + B[i, j];
    Print(C);
}
```
Parallel Program Example (Cont.)

main(argc, argv)
int argc; char *argv;
{
    Read(A);
    Read(B);
    for (p = 1; p = number-of-processors; p++)
        create-thread(p, start-procedure);
    start-procedure();
    wait-for-all-threads-to-be-done();
    Print(C);
}

start-procedure()
{
    for (i = my-rows-begin; i != my-rows-end; i++)
        for (j = 0, j ! N, j++)
        indicate-done();
}
The Parallel Programming Process**

Break up computation into tasks

Break up data into chunks

Necessary for messagepassing machines

Introduce synchronization for correctness
Synchronization

Communication – Exchange data

Synchronization – Exchange data to order events

  Mutual exclusion or atomicity
  
  Event ordering or Producer/consumer

    Point to Point
    
      Flags
    
    Global
    
      Barriers
# Mutual Exclusion

## Example

Each processor needs to occasionally update a counter.

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load reg1, Counter</td>
<td>Load reg2, Counter</td>
</tr>
<tr>
<td>reg1 = reg1 + tmp1</td>
<td>reg2 = reg2 + tmp2</td>
</tr>
<tr>
<td>Store Counter, reg1</td>
<td>Store Counter, reg2</td>
</tr>
</tbody>
</table>
Mutual Exclusion Primitives

Hardware instructions

Test&Set

Atomically tests for 0 and sets to 1

Unset is simply a store of 0

while (Test&Set(L) != 0) {;}

Critical Section

Unset(L)

Problem?
Mutual Exclusion Primitives

Hardware instructions

Test&Set

Atomically tests for 0 and sets to 1

Unset is simply a store of 0

```c
while (Test&Set(L) != 0) {};
```

Critical Section

Unset(L)

Problem - Traffic
Mutual Exclusion Primitives – Alternative?

Test&Test&Set
Mutual Exclusion Primitives – Alternative?*

Test&Test&Set

A: while (L != 0) {;
    if (Test&Set(L) == 0) {
        critical Section
    }
    else go to loop A

Problem?
Test&Test&Set

A: while (L != 0) {;
   if (Test&Set(L) == 0) {
      critical Section
   }
   else go to loop A

Problem

Traffic on lock release

What if processor swapped out while holding lock?
Mutual Exclusion Primitives – Fetch&Add

Fetch&Add(var, data)
{
  /* atomic action */
  temp = var
  var = temp + data
}
return temp

E.g., let X = 57
  P1: a = Fetch&Add(X,3)
  P2: b = Fetch&Add(X,5)
  If P1 before P2, ?
  If P2 before P1, ?
  If P1, P2 concurrent ?
Point to Point Event Ordering

Example

Producer wants to indicate to consumer that data is ready

Processor 1 | Processor 2
---|---
... | ...
A[n] = ... | ... = A[n]
Example

Producer wants to indicate to consumer that data is ready

Processor 1
A[1] = ...
A[2] = ...
...
A[n] = ...

Processor 2
while (Flag != 1) {;
...
...
...
...
...

Flag = 1
Global Event Ordering – Barriers

Example

All processors produce some data
Want to tell all processors that it is ready
In next phase, all processors consume data produced previously

*Use barriers*
Simple barrier

```c
  temp = Fetch&Inc(count)
  while (count != N) {;}
```

Problem:
Implementing Barriers**

Simple barrier

\[
\text{temp} = \text{Fetch}\&\text{Inc}(\text{count})
\]

\[
\text{while (count} \neq N) \{;\}
\]

Problem: Cannot use it again
local_flag = !local_flag
if Fetch&Inc(count) == N {
    count = 1
    flag = local_flag
}
while (flag != local_flag) {;;}