Parallel Design

Models and Paradigms
Goals & Outline

- Outline the design process of a parallel program.
- Introduce metrics for judging performance.
- Build a vocabulary of parallel programming.
- Show basic design patterns.
- Present metrics used for measuring parallelism.
Implicit / Explicit Parallelism

• Implicit: Parallelism as a result of language design, or by way of a compiler, which is transparent to the programmer.
  + Programmers do not worry about communication or task division.
  - Less than optimal code, harder to debug.

• Explicit: Parallelism by way of deliberate language constructs or annotation on top of existing languages. The programmer specifies where and when parallel constructs take place.
  + Potentially very efficient code.
  - Unique parallel bugs (ie. deadlock), longer development time.
Types of Parallelism

• Hardware
  • Instruction Level
  • Thread Level
    • Shared Memory/Cache (SMP)
  • Cluster Level
    • Message Passing

• Software
  • Task Parallelism
  • Data Parallelism
  • Hybrid Task/Data
Flynn Taxonomy of Parallelism

Single Instruction
- SISD
- SIMD

Multiple Instruction
- MISD
- MIMD
• SISD: Standard single core CPU.
• SIMD: Standard GPU processing model.
• MISD: Not commonly found in practice.
• MIMD: Standard Multi-Core model.
Target Hardware: Cluster Computing

NODE

NODE

CORE
Multi-Core CPU

CORE
Multi-Core CPU

NODE
Symmetric Multiprocessor (SMP)
Shared Memory/SMP

- Global address space for intuitive memory access.
  - Synchronization can be a tricky concept to grasp.
- Data sharing is fast, also consistent in UMA* systems.
- Memory consistency model.
  - Cache coherence.
- Lacks scalability.
  - Have to wait for hardware advances.
  - More CPUs = more memory traffic.

* UMA: Uniform Memory Access, NUMA: Non-Uniform Memory Access
Message Passing Interface (MPI)

• Distributed memory model (will run on shared memory systems).
  • Memory addresses are not mapped.
  • No globally accessible memory.
  • Hybrid systems will also use threads.

• Memory is local & scalable.
  • No need for local memory synchronization.

• You may require specialized data structures.

• Non-Uniform memory access times on remote nodes.
  • Access times affected by the network (could be Ethernet).
Challenges

- non-determinism
- communication
- synchronization
- data/task partitioning
Challenges

• non-determinism
  • race conditions
• communication
• synchronization

```java
transfer(Account from, Account to, double amount) {
    from = from - amount;
    to = to + amount;
}
```
Challenges

- non-determinism
- race conditions
- communication
- synchronization

```java
transfer(Account from, Account to, double amount) {
    temp = from;
    temp -= amount
    from = temp;

    temp = to;
    temp -= amount
    to = temp;
}

MyAccount = 500;
ClerkA.transfer(B, MyAccount, 200);
ClearB.transfer(MyAccount, C, 50);
```
Challenges

- non-determinism
- race conditions
- communication
- synchronization

<table>
<thead>
<tr>
<th>ClerkA.transfer (B, MyAccount, 200);</th>
<th>ClearB.transfer (MyAccount, C, 50);</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp = $300 (temp = from)</td>
<td>temp = $500 (temp = from)</td>
</tr>
<tr>
<td>temp = $100 (temp -= amount)</td>
<td>temp = $450 (temp -= amount)</td>
</tr>
<tr>
<td>B = $100 (from = temp)</td>
<td>=== delay ===</td>
</tr>
<tr>
<td>temp = $500 (temp = to)</td>
<td>...</td>
</tr>
<tr>
<td>temp = $700$ (temp += amount)</td>
<td>...</td>
</tr>
<tr>
<td>MyAccount = $700 (to = temp)</td>
<td>...</td>
</tr>
<tr>
<td>MyAccount = $450 (from = temp)</td>
<td></td>
</tr>
</tbody>
</table>
Challenges

- non-determinism
- communication
  - mutual exclusion (mutex)
- synchronization
Challenges

- non-determinism
- communication
  - mutual exclusion (mutex)
- synchronization

```cpp
foo()
{
  do stuff before
  lock();
  ...
  critical code section
  ...
  release();
  do stuff after
}
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>do stuff before</td>
<td>do stuff before</td>
</tr>
<tr>
<td>lock</td>
<td>lock</td>
</tr>
<tr>
<td>critical code</td>
<td>...</td>
</tr>
<tr>
<td>release</td>
<td>...</td>
</tr>
<tr>
<td>do stuff after</td>
<td>critical code</td>
</tr>
<tr>
<td></td>
<td>release</td>
</tr>
<tr>
<td></td>
<td>do stuff after</td>
</tr>
</tbody>
</table>
Challenges

• non-determinism
• communication
• synchronization
  • deadlocks

```java
transfer(Account from, Account to, double amount) {
    sync(from);
    sync(to);
    from.withdraw(amount);
    to.deposit(amount);
    release(to);
    release(from);
}
Thread1 -> transfer(A, B, 10.0);
Thread2 -> transfer(B, A, 10.0);
```
Performance Metrics

• Serial Runtime : $T_1$
• Parallel Runtime at n Processes: $T_n$
• Speedup (times faster): $S_n = T_1 / T_n$
• Efficiency: $E_n = S_n / n$
Performance Metrics: Presentation

- Serial Runtime: $T_1$
- Parallel Runtime at $n$ Processes: $T_n$
- Speedup (times faster): $S_n = T_1 / T_n$
- Efficiency: $E_n = S_n / n$
Performance Metrics: Presentation

- Serial Runtime: $T_1$
- Parallel Runtime at n Processes: $T_n$
- **Speedup (times faster):** $S_n = \frac{T_1}{T_n}$
- Efficiency: $E_n = S_n / n$
Performance Metrics: Presentation

- Serial Runtime: $T_1$
- Parallel Runtime at $n$ Processes: $T_n$
- Speedup (times faster): $S_n = T_1 / T_n$
- Efficiency: $E_n = S_n / n$
Amdahl's law

Percentage of the program which can be parallelized: \( p \)
Percentage of the program which is serial only: \( 1 - p \)
Serial runtime: \( T_1 = (1-p) T_1 + p T_1 \)
Theoretical parallel runtime: \( T_n = (1-p) T_1 + p T_1 / n \)
\( T_n / T_1 = (1-p) + p/n \)
\( S_n = T_1 / T_n = 1 / ((1-p) + pn) \) note: Amdahl calls this 'speedup in latency'.
Amdahl's law

States that the minimum execution time of a parallel program is dictated by the execution time of the serial portion of the program.

- disk I/O
- inter-process communication
- critical code segments
- lock overhead
- context switching
- latency \((p)\) can change in ratio to the number of processes.
Patterns

- Master-Worker
- Multi-Walk
- Pipeline
- Hybrid
- Loop Parallelism
Master-Worker

- **MASTER**
  - Assigns Tasks

- **WORKER**
  - Performs Computation

- **MASTER**
  - Collects Results
• All communication is between the master and a worker.
• The master can either wait (block) or perform computation.
• Scalability (see Hadoop).
• Simple to code.
• No inter-worker communication.
• Single point of failure.
Multi-Walk (single program multiple data)
• Communication is between processes, as opposed to facilitated such as in Master-Worker.
• Barrier points (communication).
• Not suited for large variation in process runtime.
• Prone to communication delays.
Pipeline
• Data assembly line model.
• Queue driven model.
  • Prone to starvation.
  • Variable sized process pools.
• Instruction pipeline.
• Graphics pipeline.
Loop Parallelism
• Data independence between iterations of the loop.
• Easy to implement.
• Susceptible to race conditions from mutex locks on critical code sections.
Hybrid Patterns (nested, asynchronous): A composition of patterns resulting in a hierarchy of tasks which allows sub-patterns to be replaced with another pattern with matching input-output dependencies.
Thank You. Questions?