Chapter 6

Parallel Processing

Evolution of parallel hardware

• I/O channels and DMA
• Pipelined functional units
• Vector processors (ILLIAV IV was built in 1974)

• Multiprocessors (cm* and c.mmp were built in the 70’s)
• Instruction pipelining and superscalers

• Supercomputers - Massively Parallel Processors (Connection machine, T3E, Blue Gene, …)
• Symmetric Multiprocessors (SMPs)

• Distributed computing (Clusters, server farms, grids, clouds)

• Multi-core processors and Chip Multiprocessors
• Graphics Processor Units (GPU) as accelerators
Pipelining and Instruction Level Parallelism

- Pipelining overlaps various stages of instruction execution to achieve performance.
- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
- Pipelining, however, has several limitations.
  - The speed of a pipeline is limited by the slowest stage.
  - Data and structural dependencies
  - Control dependencies
- One way of alleviating these limitations is to use multiple pipelines → VLIW and Superscalers

Superscalar Execution

- **In-order execution**: instructions are issued to the pipeline(s) in the order in which they are encountered. That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle.
- **Out-of-order execution**: a more aggressive model where instructions can be issued to the pipeline(s) out of order. In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue.
- Usually, cannot keep all pipelines busy all the time
Exploring System Level Parallelism

• Why?
  – ILP (Instruction Level Parallelism) is limited
  – Power consumption limits the increase in clock frequency
• Multi-tasking:
  – Run multiple tasks on separate CPUs
  – Divide your task into multiple sub-tasks, and run sub-tasks on multiple CPUs.
  – Multi-threading is a form of multi-tasking (threads are light weight tasks).
• The number of tasks (threads) does not have to be equal to the number of CPU’s – can multiplex tasks (threads) on a CPU.
• Generation of multiple tasks (threads):
  – Automatically (for example, by the compiler)
  – Specified by the user (user needs to think parallel)

How to create parallel applications

(From Mattson "Gentle Intro. to Parallel Programming")
Multiprocessors

- **Idea:** create powerful computers by connecting many smaller ones
  - **good news:** it works
  - **bad news:** it is hard to write good concurrent programs.

- **Now,** we have to deal with parallelism because of Chip Multiprocessors

- Speedup and efficiency (Section 6.2)
  - For a given problem $A$, of size $n$, let $T_p(n)$ be the execution time on $p$ processors, and $T_1(n)$ be the execution time (of the best algorithm for $A$) on one processor. Then,
    \[ S_p(n) = \frac{T_1(n)}{T_p(n)} \]
    
    Efficiency $E_p(n) = \frac{S_p(n)}{p}$
    
    Speedup is between 0 and $p$, and efficiency is between 0 and 1.
  
  - Linear Speedup means that $S$ is linear with $p$ (linearly scalable systems)
    
    - If speedup is independent of $n$, then the algorithm is said to be **strongly scalable**.
    - Otherwise, it is **weakly scalable**

SMP - Symmetric multiprocessors

Network connected MP
**Speedup and efficiency**

**Minsky’s conjecture:**
Speedup is logarithmic in $p$

**Amdahl’s law:**
If $f$ is the fraction of the task that can be executed in parallel

\[ T_p = (1-f) \cdot T_s + f \cdot T_s / p \]

Speedup $S_p = \frac{1}{(1-f) + \frac{f}{p}} \quad \text{[if } p \text{ is very large]} = \frac{1}{1-f}$

---

**Scaling Example**

- Add two 10×10 matrices then sum the diagonals of the result
  - The addition of two matrices can be done in parallel
  - The summation of 10 diagonal elements cannot be done in parallel
  - Hence, $f = 100/110$
- On a single processor: $T_s = (10 + 100) \times t_{add}$
- On 10 processors
  - $T_p = 100/10 \times t_{add} + 10 \times t_{add} = 20 \times t_{add}$
  - Speedup, $S_{10} = 110/20 = 5.5$ (55% efficiency)
- On 100 processors
  - Time = $100/100 \times t_{add} + 10 \times t_{add} = 11 \times t_{add}$
  - Speedup, $S_{100} = 110/11 = 10$ (10% efficiency)
- The above calculations assume that the load can be balanced across processors
Scaling Example (cont)

- What if matrix size is $100 \times 100$ (fixed size)?
  - Single processor: $T_s = (10000 + 100) \times t_{add}$
  - $p = 10$ processors
    - Speedup, $S_{10} = 10100/1100 = 9.18$ (91.8% efficiency)
  - $p = 100$ processors
    - Speedup, $S_{100} = 10100/200 = 50.5$ (50.5% efficiency)
- The problem is not Strongly Scalable (given a fixed problem size)

- Consider a number of processors proportional to the problem size
  - $10\times10$ matrix, $p = 10 \rightarrow S_{10} = 5.5$ -- (55% efficiency)
  - $100\times100$ matrix, $p = 100 \rightarrow S_{100} = 50.5$ -- (50.5% efficiency)
  - In both cases, we achieve almost 50% efficiency – the problem is Weakly scalable (not strongly scalable).