CSc 553 — Principles of Compilation

35 : Parallel Computers

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

Copyright © 2011 Christian Collberg

April 20, 2011

1 The Need for Speed I

- Lets assume that we need to solve a particular problem P. We must have a solution to P before a certain time T.
- Example 1: P=forecasting tomorrow's weather, T=today. A forcast for tomorrow's weather is no good to us if it's not ready until the day after tomorrow!
- Example 2: P=cryptanalysis, T=ASAP. If the enemy sends an encrypted message Attack at dawn! we'd better crack it quickly!
- So, we'll we try to solve P using the fastest processor on the market.

2 The Need for Speed II

- If that's not fast enough, our only choice is to go with a multi-processor machine. You can use multi-processors for a variety of reasons such as increased reliability and throughput (solving lots of little problems quickly), but our aim is to solve **one** really large problem **really** fast.
- Note that when we talk about **time** we mean **wall** (or **real** or **clock**) time, not CPU time. A parallel program (one which runs on several processors) will normally use **more** CPU time than one that runs on a single processor. That's OK; all we're interested in is that the program **finishes** as soon as possible, i.e. uses a minimum amount of wall time.

3

Kinds of Parallel Computers

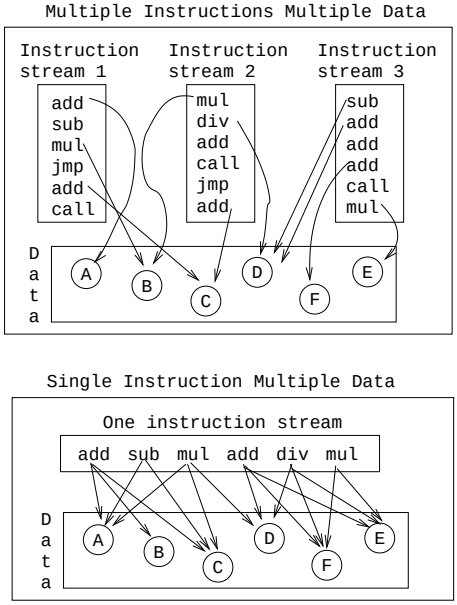
4 Kinds of Parallel Computers I

- Parallel computers are usually classified as either SIMD ("*sim-dee*") or MIMD ("*mim-dee*") and as having either **shared** or **distributed** memory.

MIMD vs. SIMD

- SIMD=Single Instruction Multiple Data. The idea is that there is **one** single stream of instructions that operate on different multiple data items. So, you may have one single `MUL`-instruction that initiates thousands of multiplications between different pairs of floating point values.
- MIMD=Multiple Instructions Multiple Data. Several independent instruction streams operate on different data items.

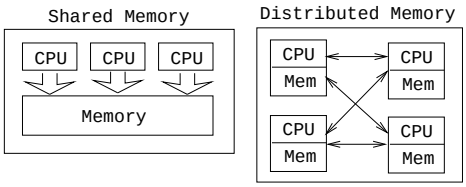
5



6 Kinds of Parallel Computers III

Shared vs. Distributed Memory

- In a shared memory machine every processor can access the entire memory. Processors can therefore communicate via shared variables.
- In a distributed memory machine each processor has its own private memory that no other processor can access. Processors communicate through message passing.



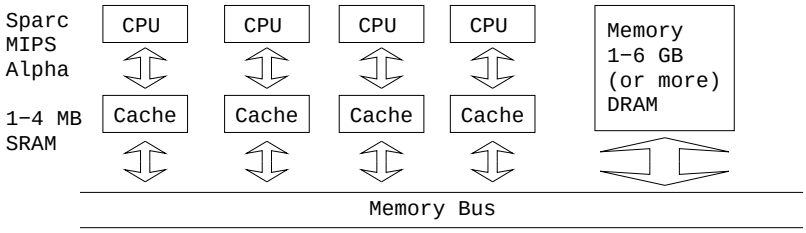
7

Shared Memory Multiprocessors

8 Shared Memory MP I

- Idea: Connect a (small) number of fast commodity microprocessors through a fast memory bus to shared memory.
- Almost all major computer vendors have such machines available.
- Accesses to shared memory becomes the major performance bottleneck. To offset this, each CPU has a large fast local cache. Still, memory bandwidth is limited, so these types of machines are usually limited to a small number of processors.

9 Shared Memory MP II



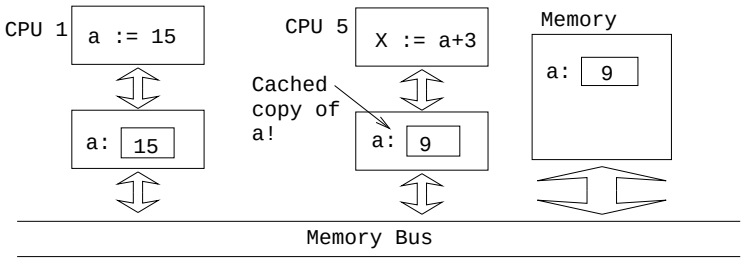
10 Cache Coherence I

- Cache coherence is also a problem. If $\texttt{CPU}_1$ writes a value into some memory location M_i, then all other CPUs must be notified of this change. If, for example, $\texttt{CPU}_2$ already has a copy of the value of M_i in its local cache, then this value must be invalidated. To solve this problem, many architectures employ **snooping caches**.
- A snooping cache continuously monitors the bus to see if some other CPU is writing to a particular memory location. All this is managed by hardware.

11 Cache Coherence II

- When writing code (or a compiler) for a shared-memory multiprocessor it is important to optimize for good cache behavior:
  - We should avoid **cache thrashing**. This occurs when two or more processors access the same memory location. The data will be pulled in and thrown out from the different caches.
  - We should avoid **cache overflows**. This occurs when we attempt to load a data structure that doesn't quite fit in the cache. Part of the structure will have to be **evicted**.

12 Cache Coherence III



- `CPU 1` assigns a new value to variable `a`. `CPU 5`'s copy of `a` is no longer valid and must be evicted from the cache.

13

Vector Machines

14 Vector Machines I

- Vector machines are built on the SIMD paradigm. The idea is that rather than operating on one scalar value at a time, we'll operate on an **array** of values. A vector machine may, for example, have an `add-vector` instruction that takes two 64-element arrays and adds them together, element by element.
- Vector machines were the first high-performance computers to be built. Although many people have predicted the demise of supercomputing in general and vector machines in particular ("the revenge of the killer micros"), vector machines are still being built today and are still some of the fastest machines around.

15 Vector Machines II

- Vector computers need very fast memory access to transfer vectors to and from the CPU. Some machines are built with fast (and expensive) SRAM as main memory, others use highly interleaved DRAM.
- Modern vector machines are combination SIMD/MIMDs; they are built as shared memory machines with custom vector processors as processing units.
- A typical vector processor has scalar as well as vector registers. Vector length is bounded, typically 64 or 128 elements. There are vector as well as scalar instructions.
- There are scalar instructions such as: `add-integer, mul-float, compare-float, load integer`, etc.

16 Vector Machines III

- The machine has arithmetic vector instructions such as these (V_i are vector registers, S_i is a scalar register):

  V_i `:=` V_j `op` V_k `op=add, mul,`$\cdots$. Vector-to-vector addition, multiplication, etc.

  V_i `:=` V_j `op` S_k `op=add, mul,`$\cdots$. Scalar-to-vector addition, multiplication, etc.

- Load and store vector instructions:

  V_i `:= memory[start:stop:incr]` Load a vector from memory into a vector register. Start loading at address `start`, end at `stop`, load every `incr` memory element.

  `memory[start:stop:incr] :=` V_i Store a vector register into memory.

17 Vector Machines IV

- Scatter/Gather vector instructions:

  V_i := memory[V_k] **Gather** words from nonsequential memory locations into register V_i: $V_i[r]$:= memory[$V_k[r]$].

  memory[V_k]:= V_i **Scatter** words from elements of register V_i to nonsequential memory locations: memory[$V_k[r]$] := $V_i[r]$.

Gather Example:

- V_1, V_2 are vector registers, A an array.
- V_1=[2,4,1,3], A=[20,10,15,99].
- V_2:=$A[V_1]$ ⇒ V_2=[10,99,20,15].

Scatter Example:

- V_1=[2,4,1,3], V_2=[20,10,15,99].
- $A[V_1]$:=V_2 ⇒ A=[100,99,20,15].

18 Vector Machines V

- Bit-operations:

  V_i := PopCount V_j $V_i[r]$:= number of bits set in $V_j[r]$.

  V_i := FirstBitSet V_j $V_i[r]$:= number of leading zeroes in $V_j[r]$.

- Other vector instructions:

  S_i := $V_j[S_k]$ Get vector element.

  $V_j[S_k]$:= S_i Set vector element.

  V_i := V_j ShiftLeft S_k Shift register V_j left S_k steps.

  V_i := Merge V_j, V_k By V_m Vector merge. $V_i[r]$:= if $V_m[r] = 1$ then $V_j[r]$ else $V_k[r]$.

  VL := C Set the vector length to C. Future vector instructions will operate on vectors of length C. C is limited by the vectore register length.

19 Vector Machines VI

Example

```
FOR i := 1 to n DO
   A[i] := B[i]
END
```

Vector Code (one processor)

```
; Assume r1 holds address of A.
; Assume r2 holds address of B.
; Assume r3 holds n.

setvl    r3          ; Set vector length.
loadv    v1, (r2)    ; Load v1 from B.
storev   v1, (r1)    ; Store v1 into A.
```

- Here we assume that `A & B` are arrays that are short enough to fit in a vector register. What do we do when they're not?

20 Vector Machines VII

- When the arrays won't fit in a vector register we have to **strip-mine**. I.e., looping over the vector, performing the vector operations repeatedly until the entire array is processed.
- Assume the vector length is 64:

Before Strip Mining

```
FOR i := 1 to 1024 DO
   A[i] := B[i]
END
```

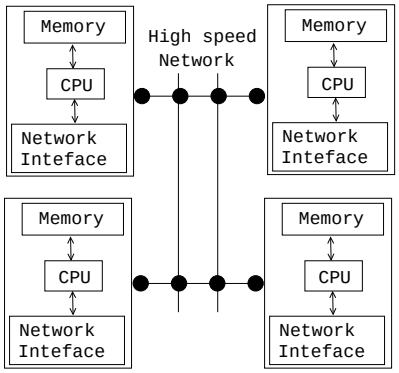
After Strip Mining

```
FOR i := 0 to 15 DO
   r1 := address of A[i*16]
   r2 := address of B[i*16]
   loadv   v1, (r2)   ; Load v1 from B.
   storev  v1, (r1)   ; Store v1 into A.
END
```

21

Distributed Memory Multiprocessors

22 Distributed Memory MPs



- The data is distributed over the processor nodes. Nodes communicate via message passing.

23 Distributed Memory MPs — Typical operations

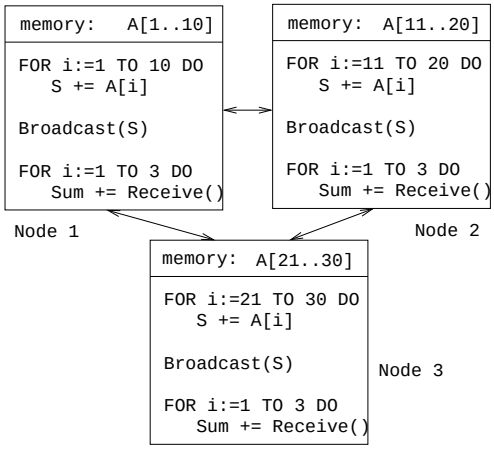
`send(P, M)` Send message (data) `M` to processor `P`.

`receive(M)` Wait to receive a message `M`.

`broadcast(M)` Send message (data) `M` to all processors.

`multicast(P, M)` Send message (data) `M` to all processors in the group `P`.

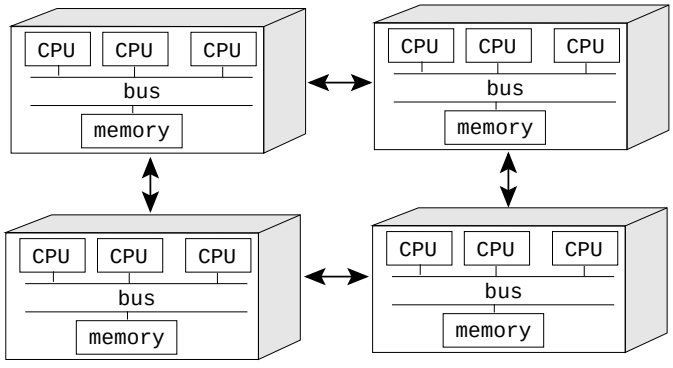
24 Example: Summing a Vector



25

The Road Ahead: SMP Clusters?

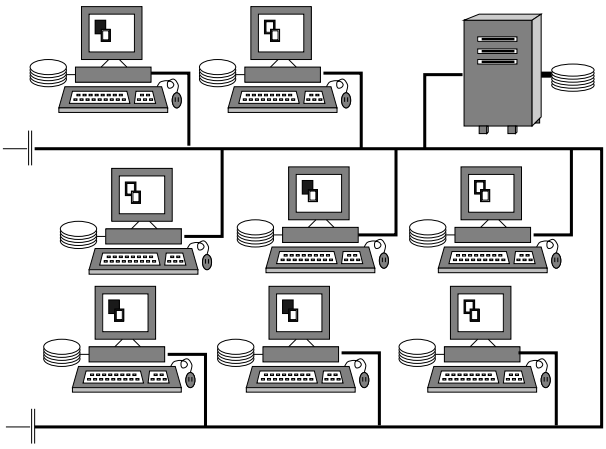
26 The Road Ahead: Shared Memory MP Clusters?



27

NOWs: Network of Workstations

28 NOWs: Network of Workstations I



29 NOWs II

- Also known as a **Workstation Farm**.
- NOWs are particularly good for **embarrassingly parallel** programs. The animated movie Toy Story, for example, was rendered on a network of SparcStations. Note that each frame can be rendered independently of the other frames!
- The advantages of NOWs are that they are cheap, readly available, and provide a lot of free cycles during off peak hours (when their regular users are asleep!).
- The disadvantages are the slow network connections.

30

Measurements

31 Analysing Performance I

$\boxed{S}$ Hardware speed in operations per second: MIPS (millions of instructions per seconds) or MFLOPS (millions of floating point operations per second).

$\boxed{p}$ Number of processors.

$\boxed{F}$ Number of operations executed by a program.

$\boxed{T}$ The time in seconds to run a program.

$\boxed{U = \frac{F}{ST}}$ The **utilization** of the machine by a program. Example: A program p_1 runs in 10 seconds, executing 10^8 instructions on a 10 MIPS machine. It's utilization is

$$U = \frac{10^8}{(10 \cdot 10^6) \times 10} = 1$$

This is an ideal (but unusual) situation.

32 Analysing Performance II

$\boxed{S_p(n) = \frac{T^*(n)}{T_p(n)}}$ We need to solve a problem P of size n. The best sequential program solving P executes in $T^*(n)$ seconds. Our parallel algorithm running on p processors takes $T_p(n)$ seconds. Then $S_p(n)$ is the **speedup**. Example: $T^*(100) = 10$, $T_{10}(100) = 1$, then

$$S_{10}(100) = \frac{T^*(100)}{T_p(100)} = \frac{10}{1} = 10$$

This is the best possible result, known as **linear speedup**. In general, we'll do worse than that.

33 Analysing Performance III

$\boxed{S'_p(n) = \frac{T_1(n)}{T_p(n)}}$ If the optimal sequential time $T^*(n)$ is unknown, we can instead measure the **self-relative speedup**. This is defined by setting $T^*(n)$ to $T_1(n)$ (the time to execute the parallel program on one processor).

$\boxed{E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{pT_p(n)}}$ The **efficiency** of a parallel program is the fraction of time that a typical processor is usefully employed. Example: $T^*(100) = 10$, $T_5(100) = 3$, $p = 5$:

$$E_5(100) = \frac{T^*(100)}{5T_5(100)} = \frac{10}{5 \times 3} = 67\%$$

34 Amdahl's Law

- Let N be the number of processors, $\mathcal{S}$ the time spent on the sequential part of the program, and $\mathcal{P}$ the time spent on the parallel part of the program. Then the speedup S_N is

$$\begin{aligned} S_N &= \frac{\texttt{Time without speedup}}{\texttt{Time with speedup}} \\ &= \frac{\mathcal{S} + \mathcal{P}}{\mathcal{S} + \frac{\mathcal{P}}{\mathcal{N}}} \end{aligned}$$

- Let 90% of the calculation be speeded up by a factor 100. The remaining 10% cannot be speeded up. Let $\mathcal{T}$ be the total time.

$$\begin{aligned} S_N &= \frac{\texttt{Time without speedup}}{\texttt{Time with speedup}} \\ &= \frac{\mathcal{T}}{0.1\mathcal{T} + \frac{0.9}{100}\mathcal{T}} = \frac{1}{0.109} < 10 \end{aligned}$$

35

Summary

36 Summary I

- What does a supercomputer do that a workstation or a PC doesn't? **It gets the job done quicker.** Late answers are wrong answers!
- There are three popular parallel architectures: shared memory multiprocessors (commodity microprocessors connected via a bus to shared memory), distributed multiprocessors (commodity microprocessors with local memory connected via a fast switch), and parallel vector computers (vector processors connected via a bus to shared memory).