Lecture 18: Multiprocessors

- Topics: multiprocessor intro and taxonomy, symmetric shared-memory multiprocessors (Sections 6.1-6.3)

Why Multiprocessors?

- Parallelism improvements in single tasks/threads yield marginal returns
- For parallelism 4N, it is more cost-effective to connect four small processors together than build a large processor with four times the width
- Applications are often inherently parallel

Taxonomy

- SISD: single instruction and single data stream: uniprocessor
- MISD: no commercial multiprocessor: imagine data going through a pipeline of execution engines
- SIMD: vector architectures: lower flexibility
- MIMD: most multiprocessors today: easy to construct with off-the-shelf computers, most flexibility

Memory Organization - I

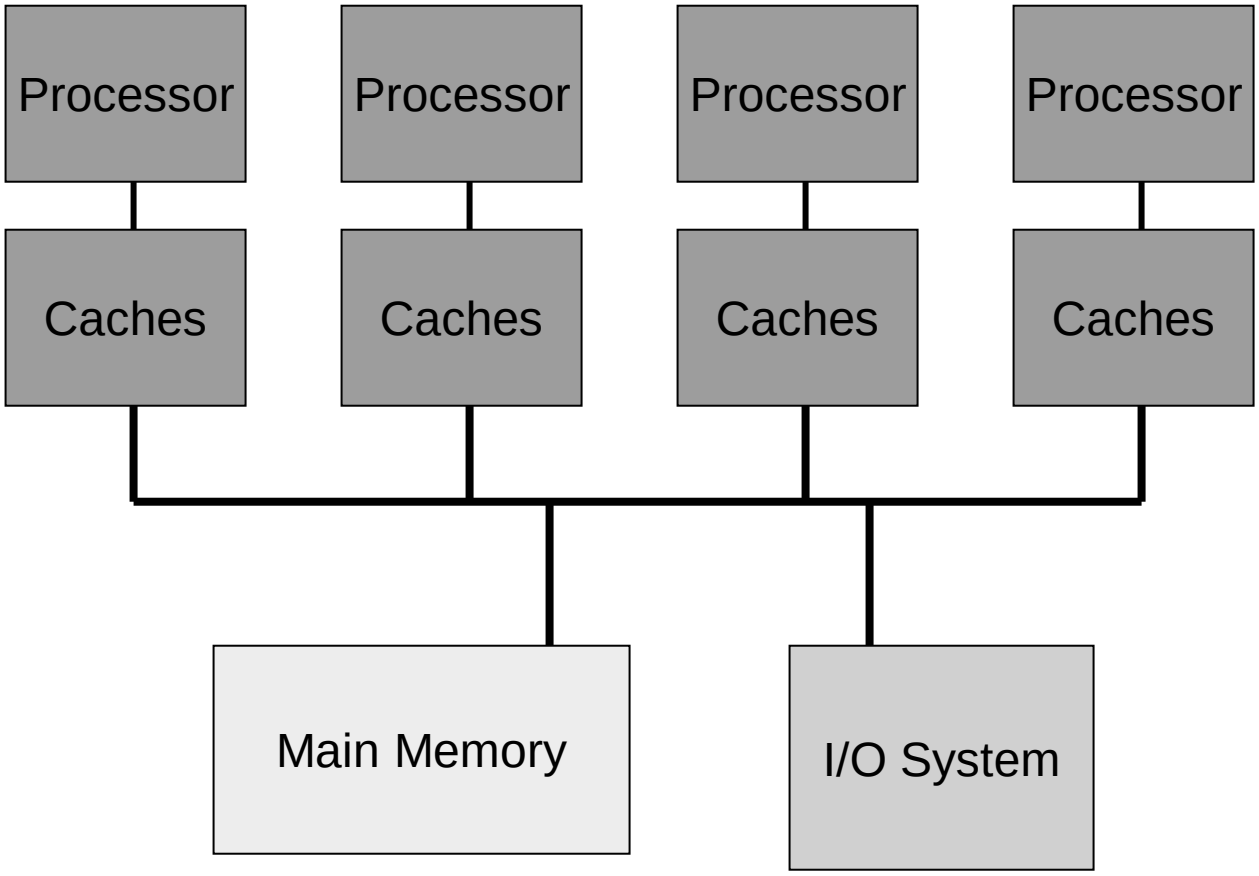
- Centralized shared-memory multiprocessor or Symmetric shared-memory multiprocessor (SMP)

- Multiple processors connected to a single centralized memory – since all processors see the same memory organization → uniform memory access (UMA)

- Shared-memory because all processors can access the entire memory address space

- Can centralized memory emerge as a bandwidth bottleneck? – not if you have large caches and employ fewer than a dozen processors

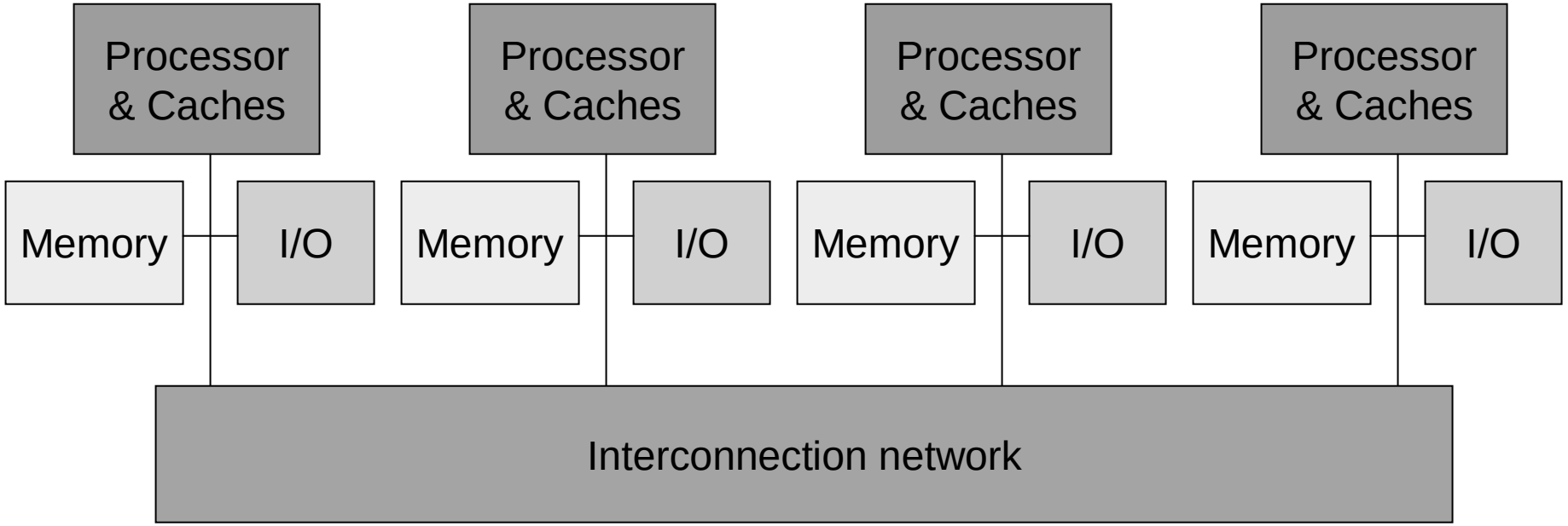
SMPs or Centralized Shared-Memory



Memory Organization - II

- For higher scalability, memory is distributed among processors → distributed memory multiprocessors
- If one processor can directly address the memory local to another processor, the address space is shared → distributed shared-memory (DSM) multiprocessor
- If memories are strictly local, we need messages to communicate data → cluster of computers or multicomputers
- Non-uniform memory architecture (NUMA) since local memory has lower latency than remote memory

Distributed Memory Multiprocessors



Shared-Memory Vs. Message-Passing

Shared-memory:

- Well-understood programming model
- Communication is implicit and hardware handles protection
- Hardware-controlled caching

Message-passing:

- No cache coherence → simpler hardware
- Explicit communication → easier for the programmer to restructure code
- Sender can initiate data transfer

Ocean Kernel

```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
      diff = 0;
      for i ← 1 to n do
        for j ← 1 to n do
          temp = A[i,j];
          A[i,j] ← 0.2 * (A[i,j] + neighbors);
          diff += abs(A[i,j] – temp);
        end for
      end for
      if (diff < TOL) then done = 1;
  end while
end procedure
```

Shared Address Space Model

```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);


main()
begin
   read(n); read(nprocs);
   A ← G_MALLOC();
   initialize (A);
   CREATE (nprocs,Solve,A);
   WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
    int i, j, pid, done=0;
    float temp, mydiff=0;
    int mymin = 1 + (pid * n/procs);
    int mymax = mymin + n/nprocs -1;
    while (!done) do
      mydiff = diff = 0;
      BARRIER(bar1,nprocs);
      for i ← mymin to mymax
        for j ← 1 to n do
          …
        endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER (bar1, nprocs);
      if (diff < TOL) then done = 1;
      BARRIER (bar1, nprocs);
    endwhile
```

Message Passing Model

```
main()
  read(n); read(nprocs);
  CREATE (nprocs-1, Solve);
  Solve();
  WAIT_FOR_END (nprocs-1);

procedure Solve()
  int i, j, pid, nn = n/nprocs, done=0;
  float temp, tempdiff, mydiff = 0;
  myA ← malloc(…)
  initialize(myA);
  while (!done) do
     mydiff = 0;
     if (pid != 0)
       SEND(&myA[1,0], n, pid-1, ROW);
     if (pid != nprocs-1)
       SEND(&myA[nn,0], n, pid+1, ROW);
     if (pid != 0)
       RECEIVE(&myA[0,0], n, pid-1, ROW);
     if (pid != nprocs-1)
       RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
     for i ← 1 to nn do
       for j ← 1 to n do
         …
       endfor
     endfor
     if (pid != 0)
       SEND(mydiff, 1, 0, DIFF);
       RECEIVE(done, 1, 0, DONE);
     else
       for i ← 1 to nprocs-1 do
         RECEIVE(tempdiff, 1, *, DIFF);
         mydiff += tempdiff;
       endfor
       if  (mydiff < TOL)  done = 1;
       for i ← 1 to nprocs-1  do
         SEND(done, 1, I, DONE);
       endfor
     endif
  endwhile
```

SMPs

- Centralized main memory and many caches → many copies of the same data
- A system is cache coherent if a read returns the most recently written value for that word

Time	Event	Value of X in	Cache-A	Cache-B	Memory
0			-	-	1
1	CPU-A reads X		1	-	1
2	CPU-B reads X		1	1	1
3	CPU-A stores 0 in X		0	1	0

Cache Coherence

A memory system is coherent if:

- P writes to X; no other processor writes to X; P reads X and receives the value previously written by P

- P1 writes to X; no other processor writes to X; sufficient time elapses; P2 reads X and receives value written by P1

- Two writes to the same location by two processors are seen in the same order by all processors – write serialization

- The memory *consistency* model defines “time elapsed” before the effect of a processor is seen by others

Cache Coherence Protocols

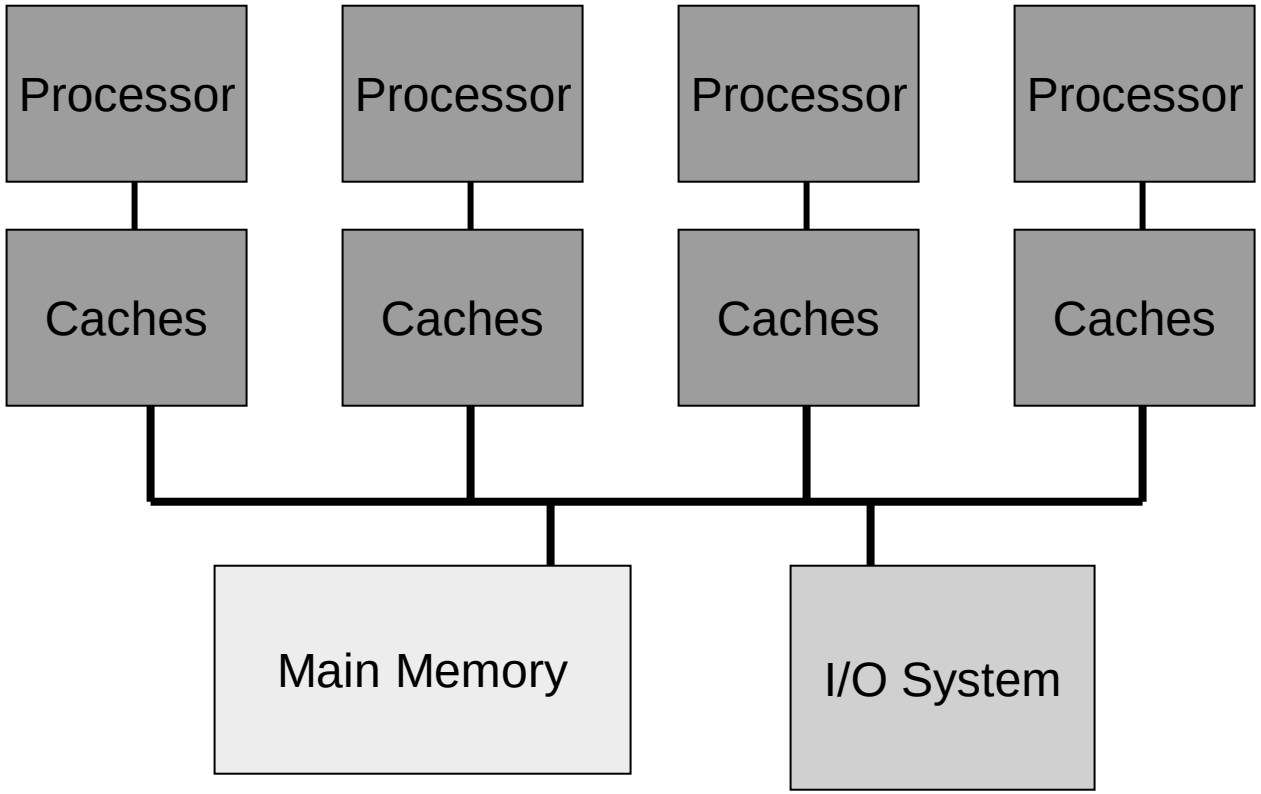
- Directory-based: A single location (directory) keeps track of the sharing status of a block of memory

- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

  - Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
  - Write-update: when a processor writes, it updates other shared copies of that block

Design Issues

- Invalidate
- Find data
- Writeback / writethrough
- Cache block states
- Contention for tags
- Enforcing write serialization



Example Protocol

Request	Source	Block state	Action
Read hit	Proc	Shared/excl	Read data in cache
Read miss	Proc	Invalid	Place read miss on bus
Read miss	Proc	Shared	Conflict miss: place read miss on bus
Read miss	Proc	Exclusive	Conflict miss: write back block, place read miss on bus
Write hit	Proc	Exclusive	Write data in cache
Write hit	Proc	Shared	Place write miss on bus
Write miss	Proc	Invalid	Place write miss on bus
Write miss	Proc	Shared	Conflict miss: place write miss on bus
Write miss	Proc	Exclusive	Conflict miss: write back, place write miss on bus
Read miss	Bus	Shared	No action; allow memory to respond
Read miss	Bus	Exclusive	Place block on bus; change to shared
Write miss	Bus	Shared	Invalidate block
Write miss	Bus	Exclusive	Write back block; change to invalid

Title

- Bullet