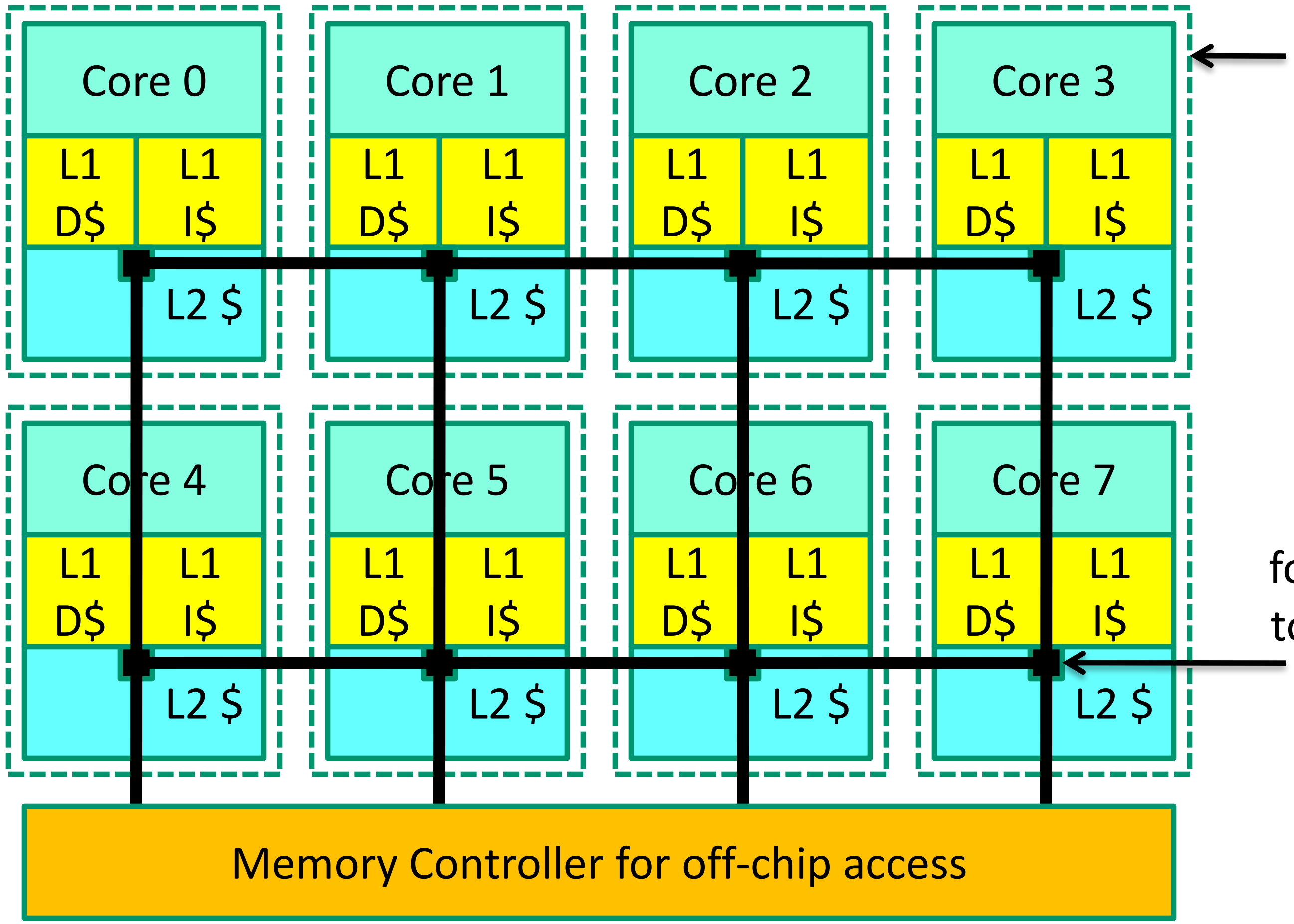
Lecture 18: Large Caches, Multiprocessors

- Today: NUCA caches, multiprocessors (Sections 4.1-4.2)

- Reminder: assignment 5 due Thursday (don’t procrastinate!)

Distributed Shared Cache

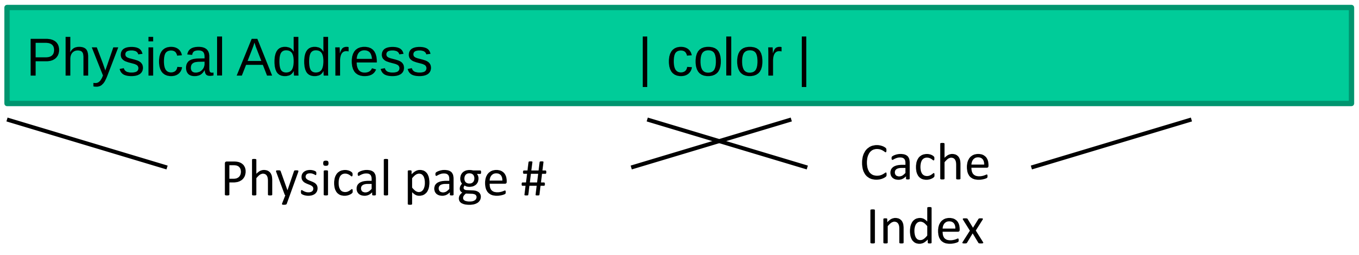


A single tile composed of a core, L1 caches, and a bank (slice) of the shared L2 cache

The cache controller forwards address requests to the appropriate L2 bank and handles coherence operations

Distributed Shared Cache

- The L2 (or L3) can be a large shared cache, but is physically partitioned into banks and distributed on chip
- Each core (tile) has one L2 cache bank adjacent to it
- One bank stores a subset of “sets” and all ways for that set
- OS-based first-touch page coloring can force a thread’s pages to have physical page numbers that map to the thread’s local L2 bank



UCA and NUCA

- The small-sized caches so far have all been uniform cache access: the latency for any access is a constant, no matter where data is found
- For a large multi-megabyte cache, it is expensive to limit access time by the worst case delay: hence, non-uniform cache architecture
- The distributed shared cache is an example of a NUCA cache: variable latency to each bank

NUCA Design Space

- Distribute Sets: Static-NUCA: Each block has a unique location; easy to find data; page coloring for locality; page migration if initial mapping is sub-optimal

- Distribute Ways: Dynamic-NUCA: More flexibility in block placement; complicated search mechanisms; blocks migrate to be closer to their accessor

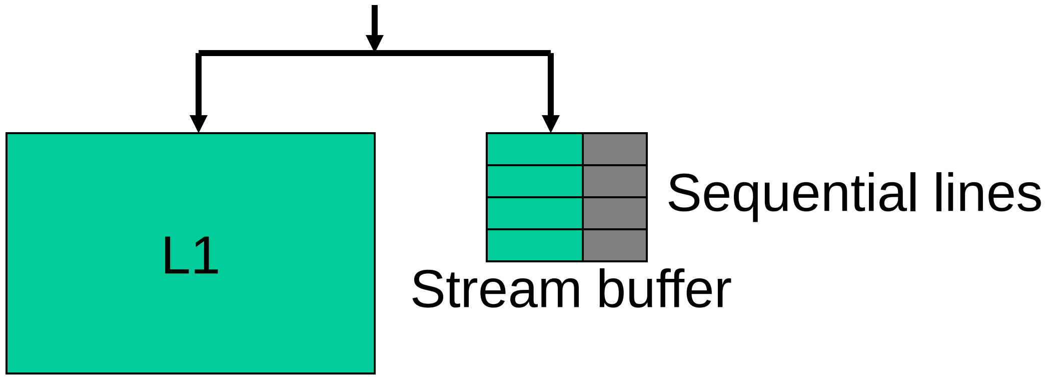
- Private data are easy to handle; Shared data must be placed at the center-of-gravity of accesses

Prefetching

- Hardware prefetching can be employed for any of the cache levels
- It can introduce cache pollution – prefetched data is often placed in a separate prefetch buffer to avoid pollution – this buffer must be looked up in parallel with the cache access
- Aggressive prefetching increases “coverage”, but leads to a reduction in “accuracy” → wasted memory bandwidth
- Prefetches must be timely: they must be issued sufficiently in advance to hide the latency, but not too early (to avoid pollution and eviction before use)

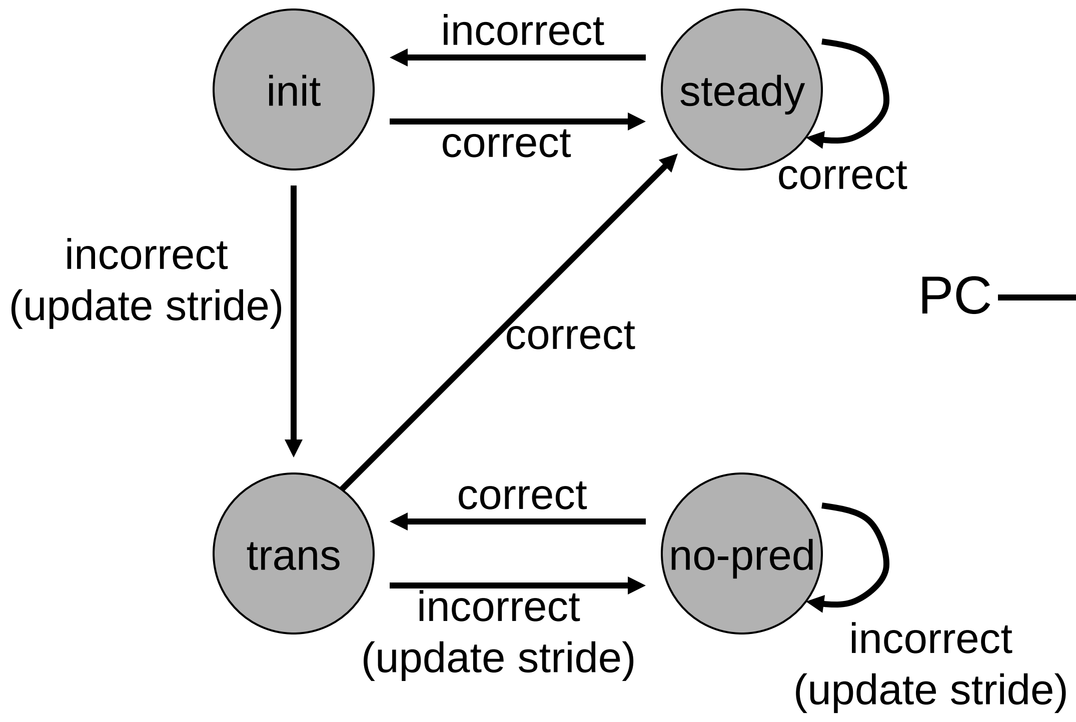
Stream Buffers

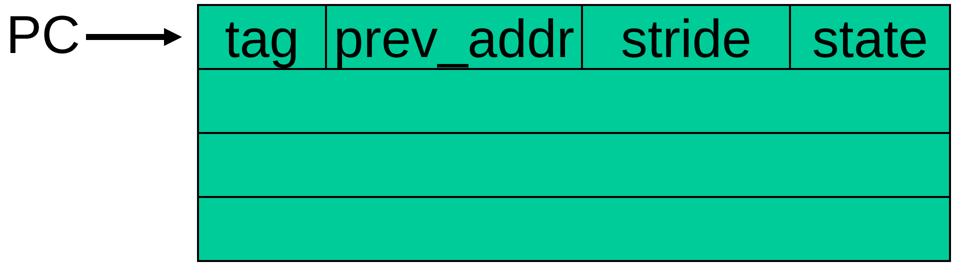
- Simplest form of prefetch: on every miss, bring in multiple cache lines
- When you read the top of the queue, bring in the next line



Stride-Based Prefetching

- For each load, keep track of the last address accessed by the load and a possibly consistent stride
- FSM detects consistent stride and issues prefetches





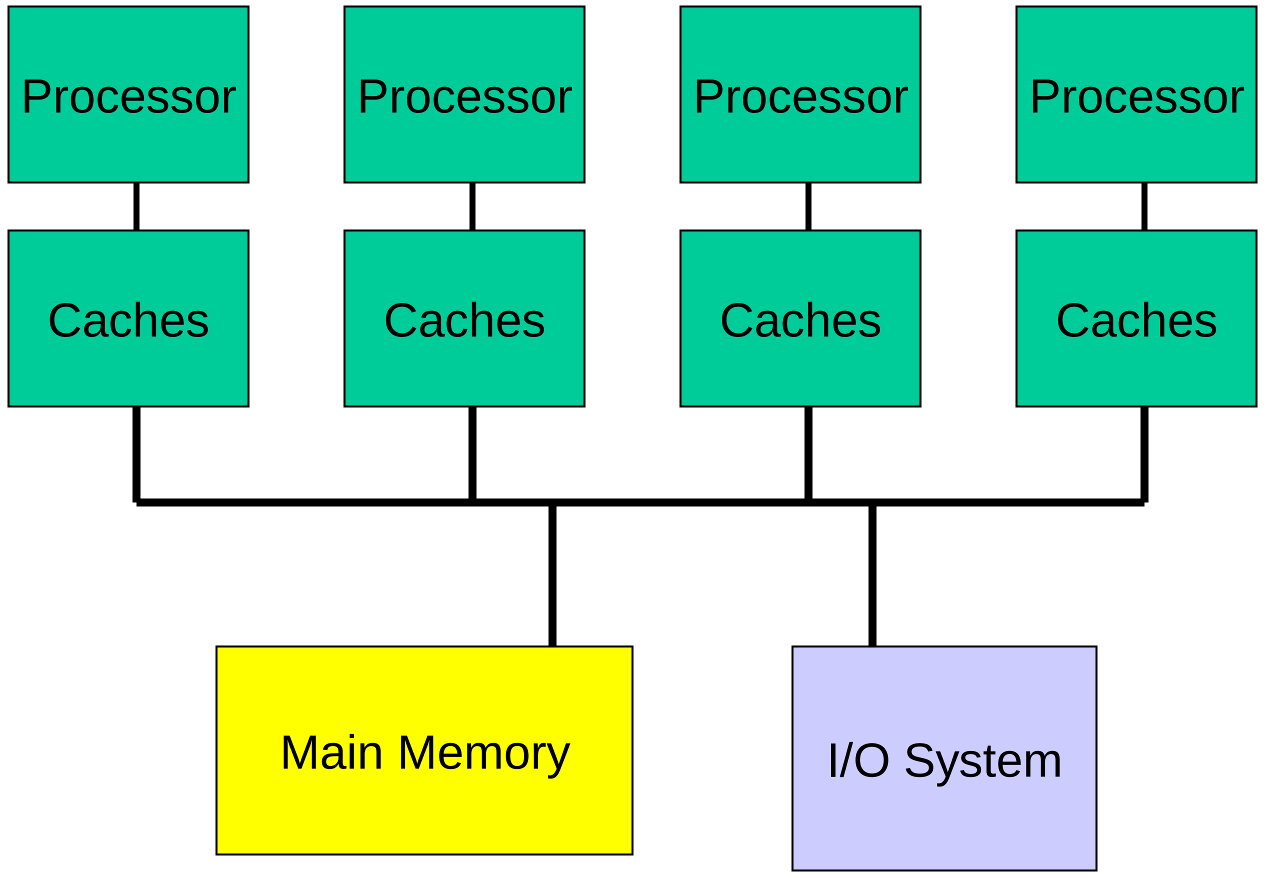
Taxonomy

- 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

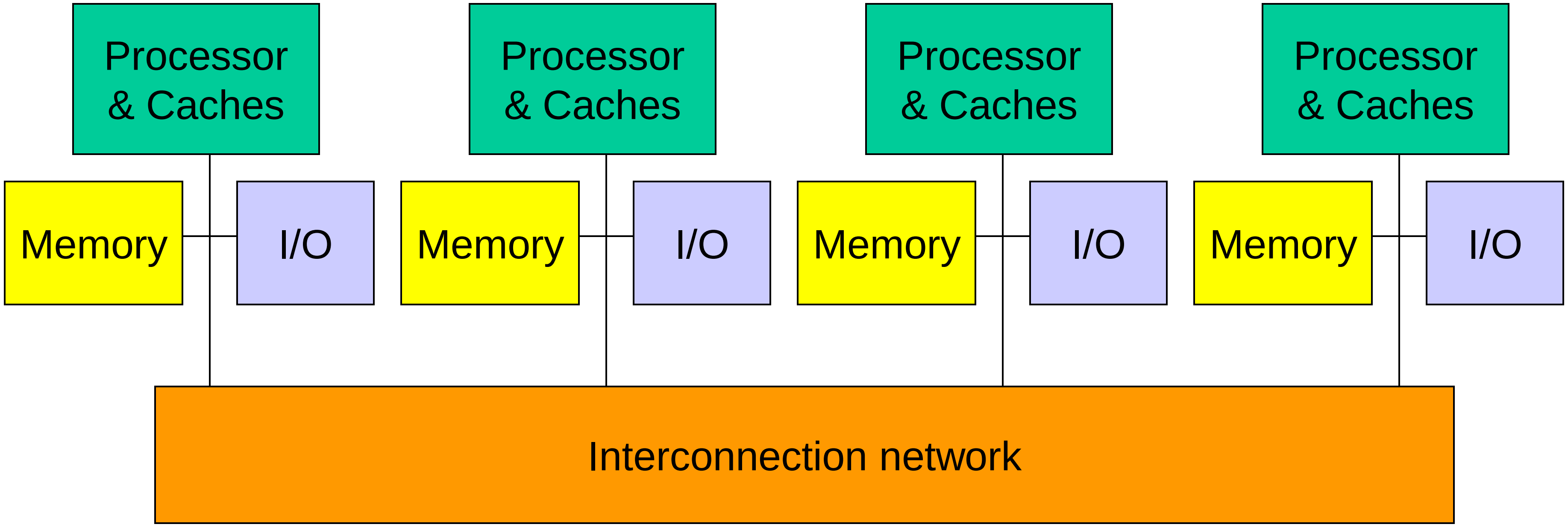
SMPs or Centralized Shared-Memory



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
```

Title

- Bullet