Lecture 1: Introduction

- Course organization:
  - 13 lectures on parallel architectures
    - ~5 lectures on cache coherence, consistency
    - ~3 lectures on TM
    - ~2 lectures on interconnection networks
    - ~2 lectures on large cache hierarchies
    - ~1 lecture on parallel algorithms
  - 10 lectures on memory systems (taught by Al)
  - 5 lectures: student presentations related to course project

Logistics

- Texts: Parallel Computer Architecture, Culler, Singh, Gupta
  Principles and Practices of Interconnection Networks, Dally & Towles
  Introduction to Parallel Algorithms and Architectures, Leighton
  Transactional Memory, Larus & Rajwar
  Memory Systems: Cache, DRAM, Disk, Jacob et al.

- Multi-threaded programming assignment due in Feb

- Final project report due in early May (will undergo conference-style peer reviewing)

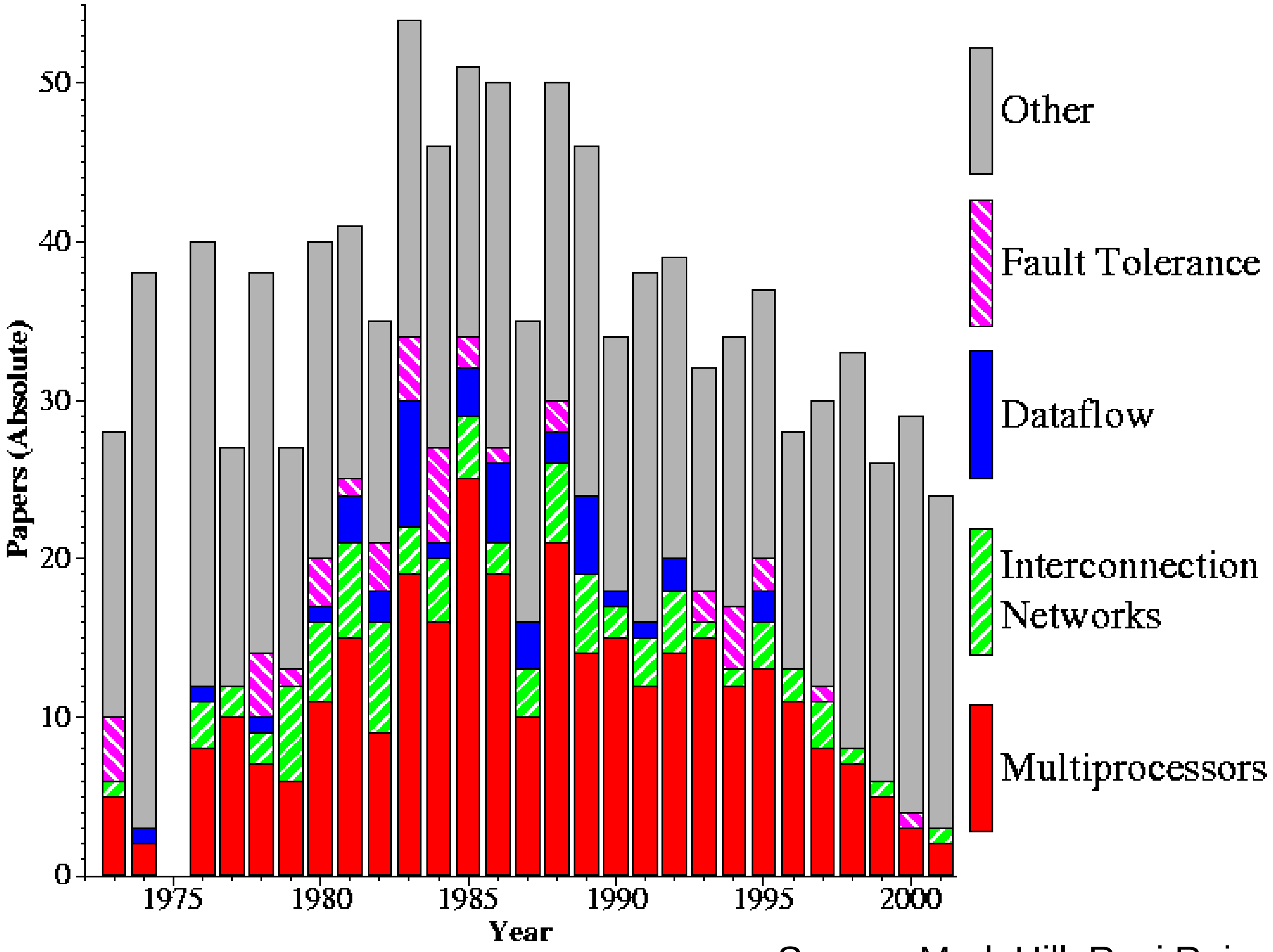
More Logistics

- Sign up for 7810 (3 credits) or for 7960 (2 credits)

- Projects: simulation-based, creative, be prepared to spend time towards end of semester – more details on simulators in a few weeks

- Grading:
  - 50% project
  - 20% multi-thread programming assignment
  - 10% paper presentation
  - 20% take-home final

Parallel Architecture Trends



ISCA papers 1973-2001

Source: Mark Hill, Ravi Rajwar

CMP/SMT Papers

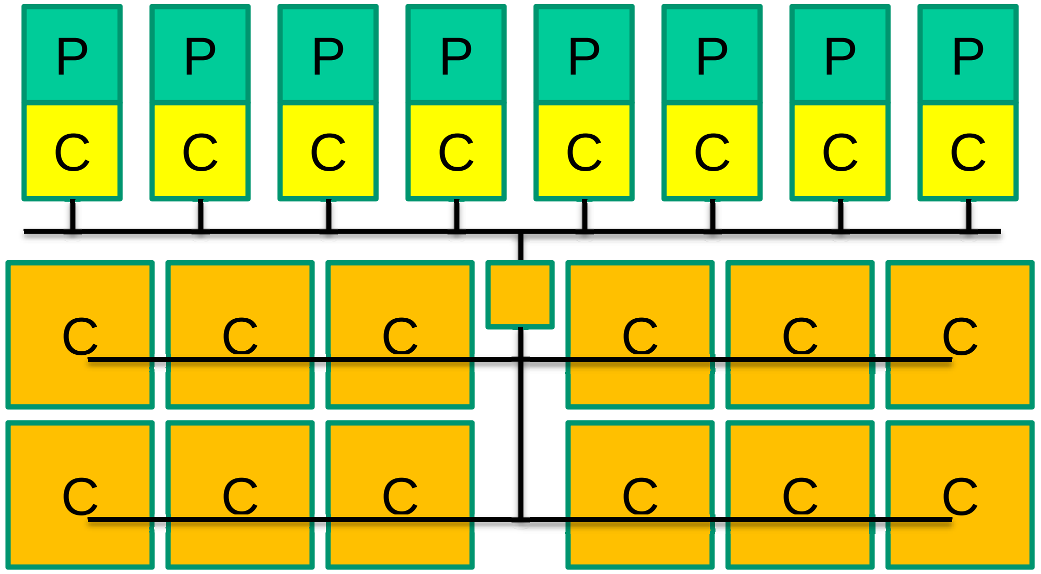
- CMP/SMT/Multiprocessor papers in recent conferences:

	2001	2002	2003	2004	2005	2006	2007
➢ ISCA:	3	5	8	6	14	17	19
➢ HPCA:	4	6	7	3	11	13	14

Bottomline

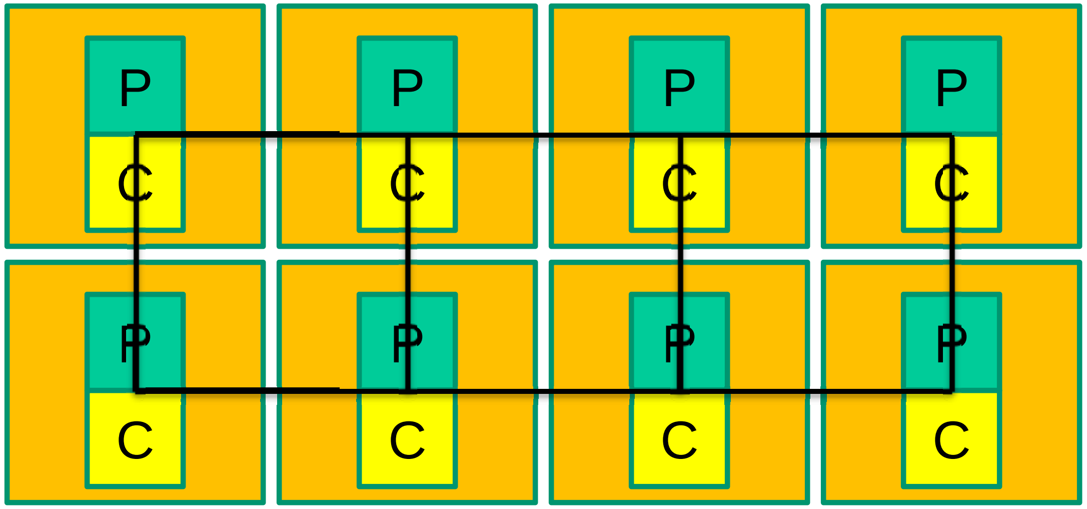
- Can’t escape multi-cores today: it is the baseline architecture
- Performance stagnates unless we learn to transform traditional applications into parallel threads
- It’s all about the data!
  Data management: distribution, coherence, consistency
- It’s also about the programming model: onus on application writer / compiler / hardware
- It’s also about managing on-chip communication

Multi-Core Cache Organizations



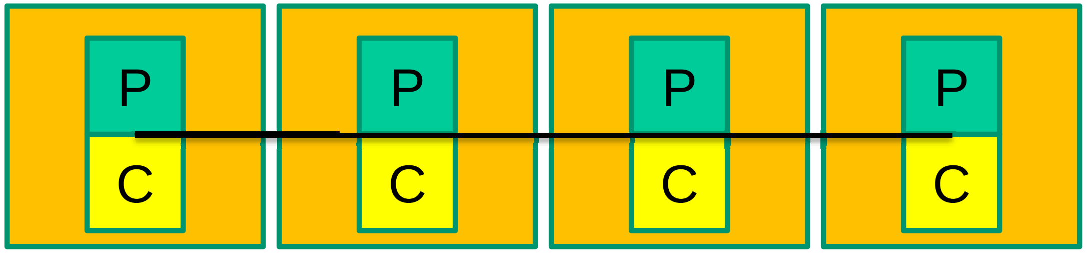
Private L1 caches
Shared L2 cache
Bus between L1s and single L2 cache controller
Snooping-based coherence between L1s

Multi-Core Cache Organizations



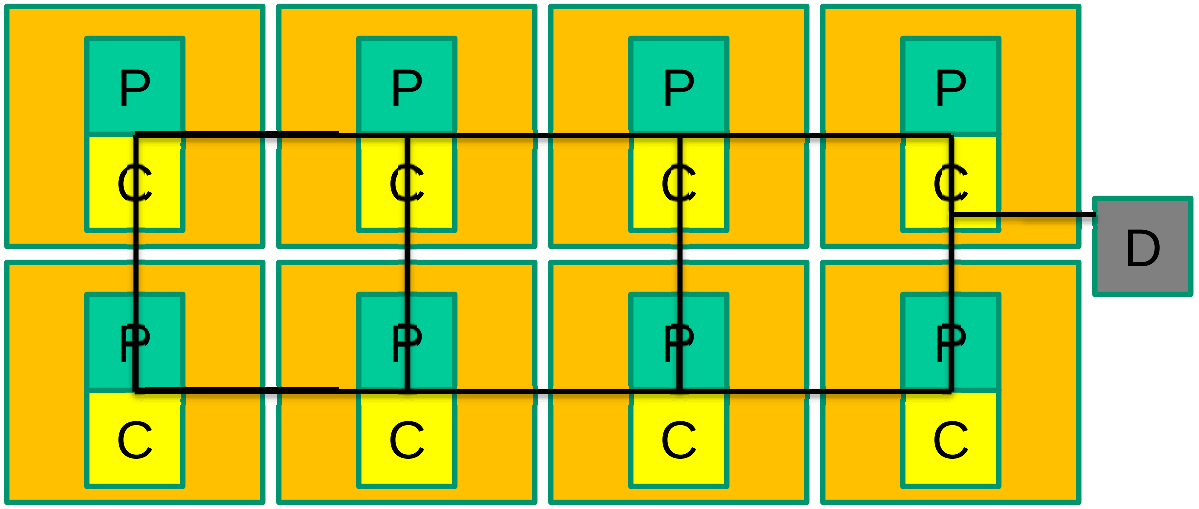
Private L1 caches
Shared L2 cache, but physically distributed
Scalable network
Directory-based coherence between L1s

Multi-Core Cache Organizations



Private L1 caches
Shared L2 cache, but physically distributed
Bus connecting the four L1s and four L2 banks
Snooping-based coherence between L1s

Multi-Core Cache Organizations



Private L1 caches
Private L2 caches
Scalable network
Directory-based coherence between L2s
 (through a separate directory)

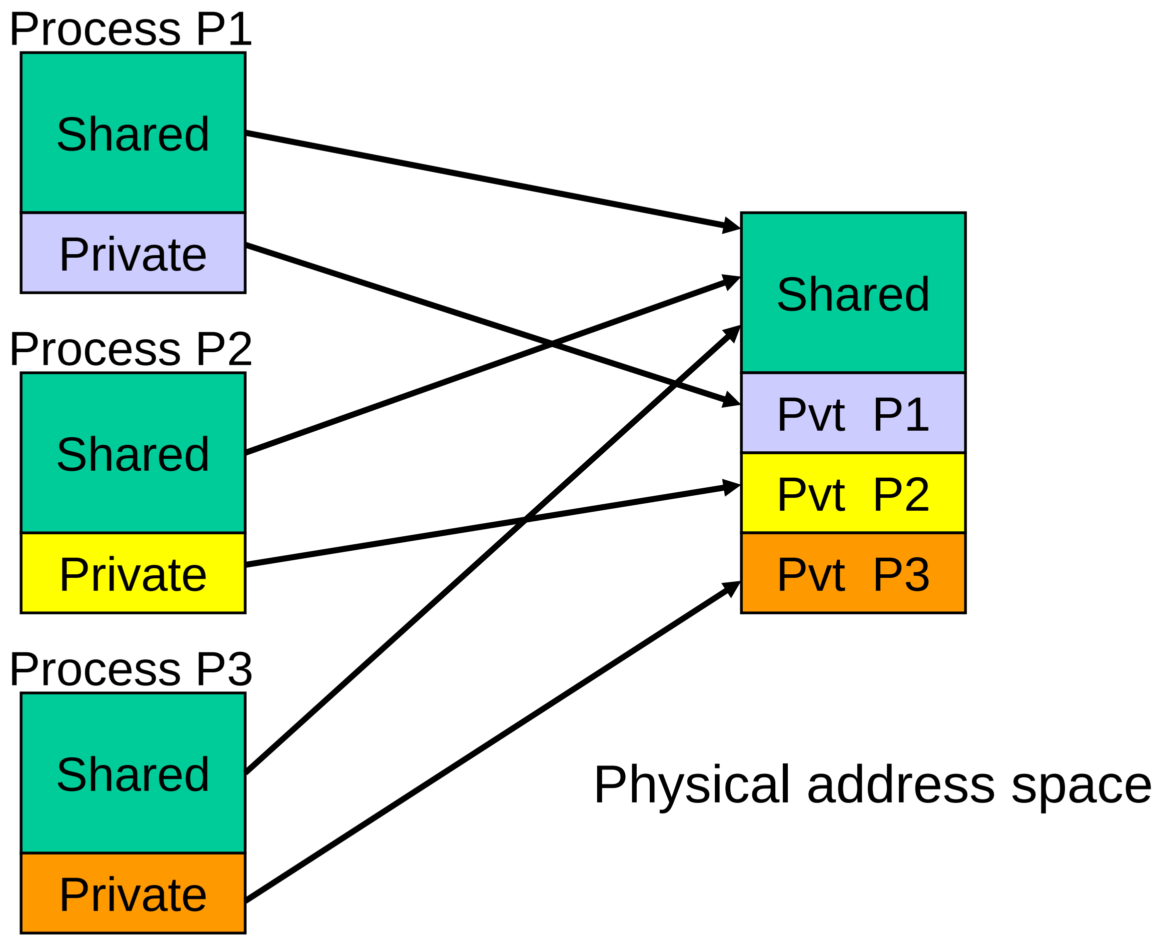
Shared-Memory Vs. Message Passing

- Shared-memory
  - single copy of (shared) data in memory
  - threads communicate by reading/writing to a shared location
- Message-passing
  - each thread has a copy of data in its own private memory that other threads cannot access
  - threads communicate by passing values with SEND/ RECEIVE message pairs

Shared Memory Architectures

- Key differentiating feature: the address space is shared, i.e., any processor can directly address any memory location and access them with load/store instructions
- Cooperation is similar to a bulletin board – a processor writes to a location and that location is visible to reads by other threads

Shared Address Space



Virtual address space
of each process

Message Passing

- Programming model that can apply to clusters of workstations, SMPs, and even a uniprocessor
- Sends and receives are used for effecting the data transfer – usually, each process ends up making a copy of data that is relevant to it
- Each process can only name local addresses, other processes, and a tag to help distinguish between multiple messages
- A send-receive match is a synchronization event – hence, we no longer need locks or barriers to co-ordinate

Models for SEND and RECEIVE

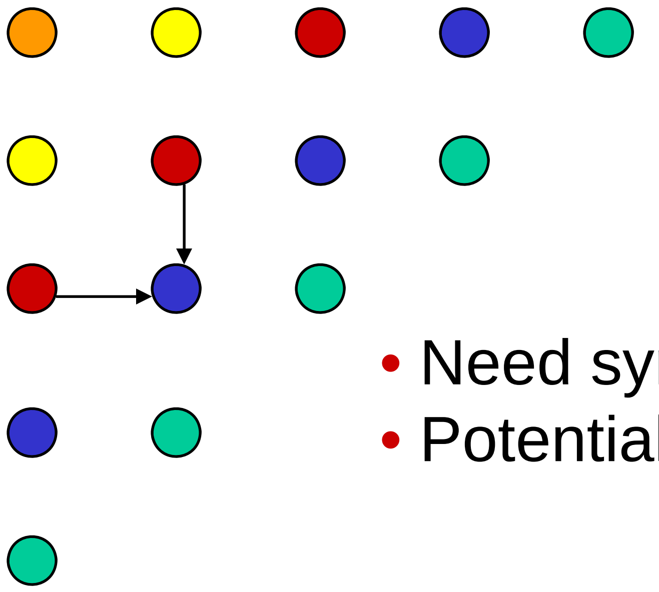
- Synchronous: SEND returns control back to the program only when the RECEIVE has completed

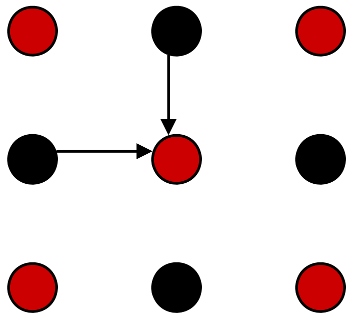
- Blocking Asynchronous: SEND returns control back to the program after the OS has copied the message into its space -- the program can now modify the sent data structure

- Nonblocking Asynchronous: SEND and RECEIVE return control immediately – the message will get copied at some point, so the process must overlap some other computation with the communication – other primitives are used to probe if the communication has finished or not

Deterministic Execution

- Shared-memory vs. message passing
- Function of the model for SEND-RECEIVE
- Function of the algorithm: diagonal, red-black ordering





- Need synch after every anti-diagonal
- Potential load imbalance

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