Lecture 5
Communication Models: Shared memory and Message Passing

Daniel Sanchez and Joel Emer

6.888 Parallel and Heterogeneous Computer Architecture
Spring 2013

Massachusetts Institute of Technology

Administrivia

- HW1 is out!
  - Due March 6
  - Code and data under MIT certs or from Stata
- Start thinking about project
  - Explore possible teams!
  - Project proposal (~2 page) due March 18
    - Ask us about topics, infrastructure, etc. *beforehand*
- Start thinking about seminars

Today's Menu

- A bit more on evaluating *parallel* systems
- Some notes on HW1
- Communication models & paper discussion

Statistically Significant Experiments

- Most fields: "Our experiment shows the vaccine is effective in 85%(+/-2%) of subjects…"
- Computer architects (often): "We ran each experiment once, here are the bars"
  - "What are the confidence intervals?" Common responses:
    - Madness is doing the same thing twice and expecting a different result!
    - Simulations take a long time! Better to simulate for 5x longer…
    - Confidence what?
- The Java tribe: "We ran each benchmark 10 times and report the best execution times"
  - The other 9 are to warm the JVM up…

Observational Error

- Most experiments (and definitely computers) are subject to variability
- Two types of observational error:
  - Systematic: Always occurs in the same way
    - Performance counter bugs, instrumentation overhead, room temperature & turbo, simulator bugs…
  - Random: Due to natural system variability and non-determinism
    - Initial machine state, VM mappings, ASLR, interrupts, benchmarks that use randomized algorithms, …
    - In parallel systems, *amplified* by lock acquisition order, barrier synchronization, etc.

- Avoiding systematic error:
  - Detect them… good luck
  - Either redesign experiment or estimate impact and adjust measurement
- Reducing random error: Make your confidence intervals small

Confidence Intervals

- We take N samples from a population (e.g., run a benchmark N times) and want to approximate a *parameter* about the whole population (e.g., the *true mean time* of all runs) with those samples (e.g., the *sample mean time* of the N runs)
  - Can we compute the actual error between both?
- An X% confidence interval is the range of values that is X% likely to contain the true value across the whole population
  - Multiple ways to estimate
  - Typically, assume gaussian distribution, compute sample mean and std, and use inverse CDF to compute (symmetric) range
  - In most real-world systems, increasing N makes interval smaller
    - Infinite-variance distributions exist, in paper…

The real system has randomness… does your simulator?

- Same starting state, no interrupts, deterministic event ordering? You have a problem
  - e.g., your benchmark executes +/-10% of instructions in the real system depending on e.g., starting machine state
  - Your baseline design happens to hit the -10%
  - Your 5% IPC-improved design happens to hit the +10%...
  - Often worse in parallel benchmarks
- Add some randomness, even artificially (+/-2 cycles on memory accesses) [Alameldeen and Wood, IEEE Micro 06]
  - May not model the real randomness, but often good enough

Sampling and cold-start effects

- Often, can only run short benchmarks (~100M instrs)
  - But want to estimate performance of much longer runs!
- Problem 1: Choose statistically significant portions of the program. Options:
  - Analyze the workload beforehand, pick samples [SimPoints, Sherwood et al, ISCA/SIGMETRICS 03]
  - Periodic or randomized sampling, and treat it as a sampling problem [SMARTS, Wunderlich et al, ISCA 03]
- Problem 2: Microarchitectural state (caches, predictors, etc) not warmed up!
  - Functional-only or detailed (timing) warming

Summarizing Performance

- Ideal world:
  - Ideal chip manufacturer: Compared to our old chip, our new one improves performance of benchmark 1 by 10%, benchmark 2 by 50%, benchmark 3 by -10%, etc.
  - Ideal customer 1: I mostly run (something similar to) benchmark 2, let's upgrade
  - Ideal customer 2: I'm half ~1, half ~3, not for me…

- Real word:
  - Customer: I don't know what I run, just give me a number!
  - Chip manufacturer: OK, here's the mean improvement…

Means

- Arithmetic: $$amean = \frac{1}{N}\sum_{i=1}^{N} x_i$$
- Harmonic: $$hmean = N \Big/ \sum_{i=1}^{N} \frac{1}{x_i}$$
- Geometric: $$gmean = \left(\prod_{i=1}^{N} x_i \right)^{1/N}$$
- For positive differing quantities, amean > gmean > hmean
- Rules of thumb: amean for absolutes, hmean for rates (speeds), gmean for ratios
- In practice, use first principles as much as possible to derive aggregate metrics
  - Weighting or other means can be useful
  - And be honest… (Q: most/least used means in papers?)

Scalability

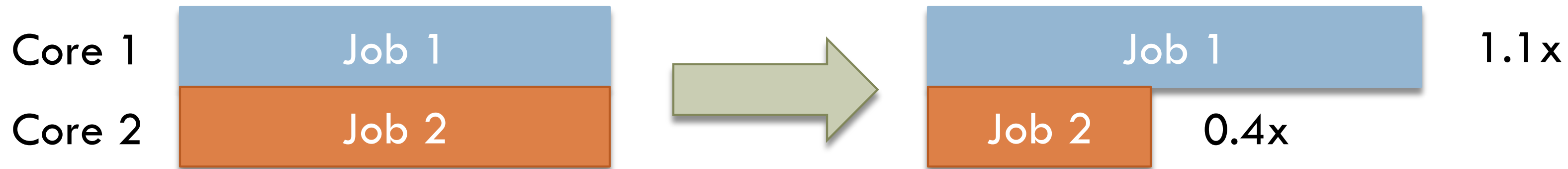
- Speedup(N) = Time on 1 processor/Time on N processors
  - What's the best we can do? Linear?
  - Often sublinear…
- Strong scaling: Speedup on 1…N processors with fixed *total* problem size
- Weak scaling: Speedup on 1…N processors with fixed *per-processor* problem size

Work != Instructions

- Especially with multithreaded benchmarks
- Classic example: Spinning
  - Increase memory latency, more spinning on lock acquires, spinning is really fast → higher IPC!
- Solution 1: Run applications to completion
- Solution 2: Instrument applications to measure units of work, measure time needed for N units
- Solution 3: Discount "useless instructions"
  - Great because we can still correlate to architectural metrics
  - But often hard in full-system simulations…

Multi-programmed setups

- Parallel processors execute multiple jobs…
- How to compute performance improvement of this?



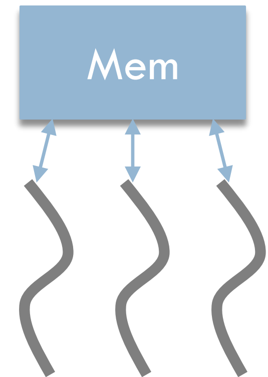
- Options (assuming work == instructions):
  - Variable-work methodology: Measure time to finish N instructions
    - Issues?
  - Fixed-work methodology: Measure time to finish N instructions for each program, then average
    - Terminate/keep running/rewind programs as they finish?
    - Issues?

HW1 Notes

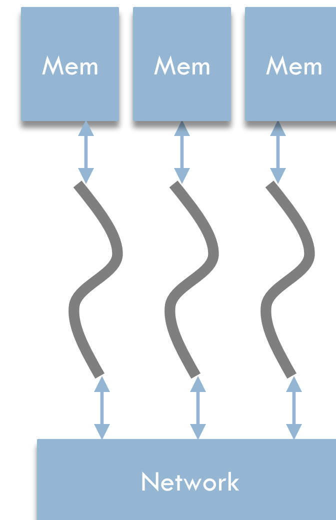
- "Here's a simulator driver and some base code, build a cache hierarchy and measure how it does"
  - Underspecified problem, on purpose
  - Very simple core & memory model (why?)
  - ST or MP workloads (SPECCPU2006), so no coherence needed
  - Some unspecified dimensions:
    - Multilevel policies: Inclusive, *non-inclusive*, exclusive
      - Hard to do inclusive as is (hint: what does inclusion require?)
    - Write-through (hard, we only give you cache line addresses) vs *write-back*
    - Set selection policy (*bit-selection* or hashing)
- Remember to use an appropriate methodology
  - Most issues are minor (work ~ instructions, minimal variability…)
  - Problem 3 explores fixed vs variable-work
  - Problem 5 requires design space exploration… don't try to bruteforce
- Questions? Mieszko, staff list

Communication Models

- Shared memory:
  - Single address space
  - Implicit communication by reading/writing memory
    - Data
    - Control (semaphores, locks, barriers, …)
  - Low-level programming model: threads (e.g., pthreads)

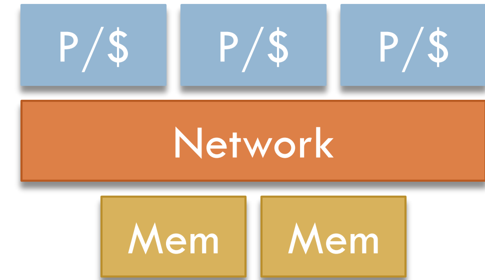


- Message passing:
  - Separate address spaces
  - Explicit communication by send/rcv messages
    - Data & control (blocking msgs, barriers, …)
  - Low-level programming model: processes + IPC (e.g., MPI)

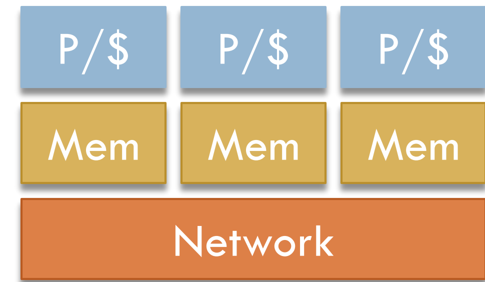


MIMD Taxonomy

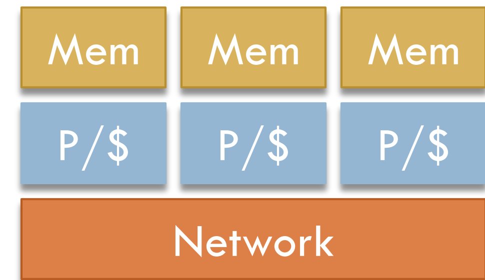
- Shared memory:
  - Uniform Memory Access (UMA): Small-scale SMPs & CMPs (e.g., P6)



  - Non-Uniform Memory Accesses (NUMA):
    - Cache-coherent (ccNUMA) (e.g., Origin, Cray T3E, modern multi-socket)
      - Cache-only (COMA) (e.g., KSR1)
    - Non-coherent (e.g., Cray T3D)



- Message-passing:
  - Massively Parallel Processors (MPPs): Tightly-coupled, high-performance parts (e.g., BlueGene/Q)
  - Clusters: Loosely coupled, commodity parts (e.g., datacenters)

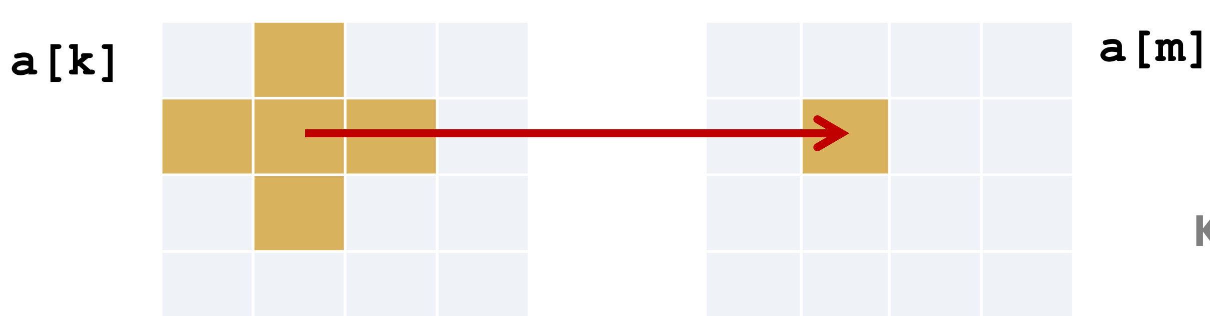


Shared Memory vs Message-Passing Programming

- Shared memory: Typically,
  - Easier to improve incrementally
    - Start with sequential version, add synchronization, analyze bottlenecks
  - Harder to fully optimize
    - False sharing, spinning, remote accesses, harder to analyze…
  - Harder to scale
    - Communication is implicit → Ignore, overuse
- Message passing: Typically,
  - Harder to improve incrementally
    - Explicit data partitioning and communication; changing algorithm often requires rewrite
  - Easier to fully optimize
    - Easier to analyze, easier to hide latencies
  - Easier to scale
    - Explicit communication is explicit → Think about it, minimize

Example: Iterative Solver

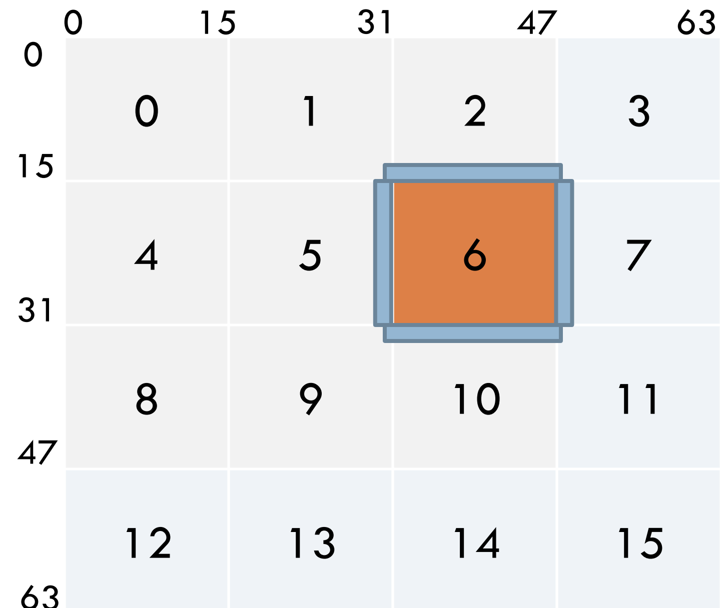

```
double a[2][MAXI+2][MAXJ+2]; //two copies of state
                             //use one to compute the other
for (s = 0; s < STEPS; s++) {
  k = s&1; // 0 1 0 1 0 1 ...
  m = k^1; // 1 0 1 0 1 0 ...
  for(i = 1; i <= MAXI; i++) {  // do iterations in parallel
     for(j=1; j <= MAXJ; j++){
            a[k][i][j] = c1*a[m][i][j] + c2*a[m][i-1][j] +
                         c3*a[m][i+1][j] + c4*a[m][i][j-1] +
                         c5*a[m][i][j+1];
     }
  }
}
```



[based on Kozyrakis & Binkert, EE282 L7, 2011]

Data Partitioning & Communication

- Divide matrix in square blocks
  - e.g. 64x64 matrix, each processor owns a 16x16 submatrix
- Processor 6
  - Owns [i][j] = [32…47][16…31]
  - Shares [i][j] = [31][16…31] and three other strips
- Each processor:
  - Communicates to get shared data it needs
  - Computes its data



Message-Passing Code


```
Fork N processes and distribute subarrays to processors
 Each process computes north[p], south[p],
 east[p], west[p],-1 if no neighbor in direction

for (s=0; s<STEPS; s++) {
    k = s&1;
    m = k^1;
    if (north[p]>= 0) send(north[p], NORTH, a[m][1][1..MAXSUBJ]);
    if (east[p]>= 0) send(east[p], EAST, a[m][1..MAXSUBI][1]);
    same for south and west
    if (north[p]>= 0) receive(NORTH, a[m][0][1..MAXSUBJ]);
    same for other directions
    for (i=1; i<=MAXSUBI; i++) {
          for (j=1; j<=MAXSUBJ; j++){
                    a[k][i][j] =  c1*a[m][i][j] + c2*a[m][i-1][j] +
                                        c3*a[m][i+1][j] + c4*a[m][i][j-1] +
                                        c5*a[m][i][j+1];
          }
    }
}
```

Shared Memory Code


```
Create N threads
 Each thread p computes istart[p], iend[p], jstart[p], jend[p]
 Each thread runs:
for (s=0; s<STEPS; s++) {
   k = s&1;
   m = k^1;
   for(i=istart[p]; i<=iend[p]; i++) {       // e.g. 32..47
        for(j=jstart[p]; j<=jend[p]; j++){  // e.g. 16..31
               a[k][i][j] =  c1*a[m][i][j] + c2*a[m][i-1][j] +
                             c3*a[m][i+1][j] + c4*a[m][i][j-1] +
                             c5*a[m][i][j+1];
        }
   }
   barrier();
}
```

So much easier! And similar performance!

And no one would have written it this way first!

The Perils of Implicit Communication

- By writing MP version first, we forced ourselves to think about data partitioning and communication
- Most shared mem programmers just do this:

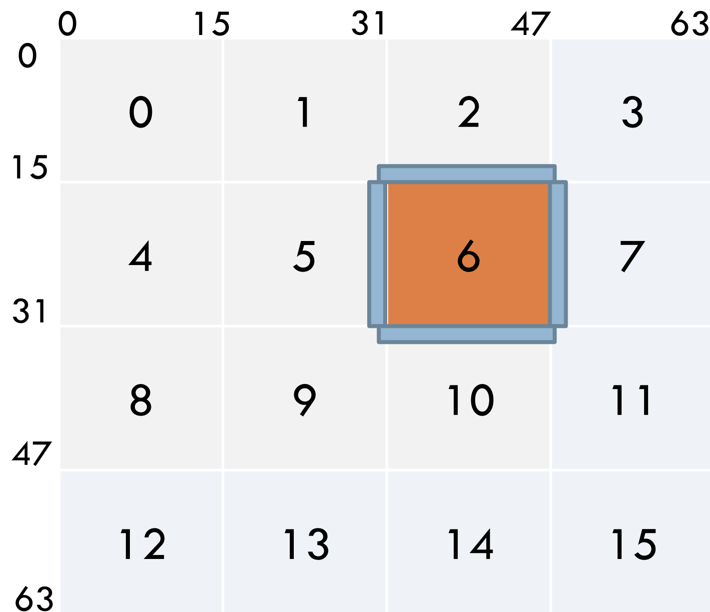
```
for(i=istart[p]; i<=iend[p]; i++) {
      for(j=start; j<=end; j++){
```

- High-level programming models are good, right?

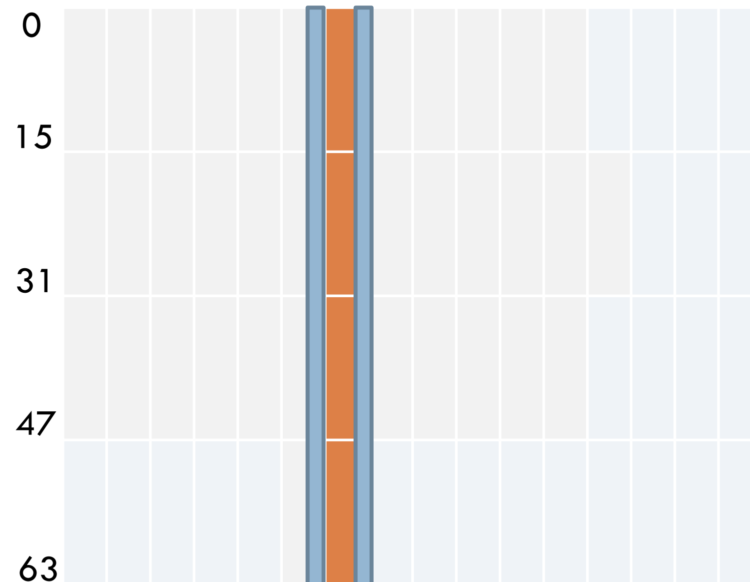
```
#pragma omp parallel for
for(i=istart; i<=iend; i++) {
      for(j=jstart; j<=jend; j++){

forall(i=istart; i<=iend; i++) {
      for(j=jstart; j<=jend; j++){
```

- What's the issue here?

Computation/Communication Ratio



Send 16*4 elems
Rcv 16*4 elems
Compute 16*16 elems



Send 64*2 elems
Rcv 64*2 elems
Compute 64*4 elems

- Uh-oh… 2x communication/computation ratio
  - How does it scale to larger matrices/processor counts?

Shared Memory Discussion

- UMA Scalability?
- NUMA Scalability?
- Cache coherence, consistency, atomic operations
  - Complexity?
  - Alternatives?

- Can we cheaply emulate message-passing on shared memory HW?

Message-passing Discussion

- Network speed/latency
  - Memory bus vs I/O bus
- Messaging overheads: Buffering, copying, protection
  - OS-level vs user-level messaging
  - Protocol overheads vs network complexity
- Synchronization overheads: Synchronous vs asynchronous
  - Polling vs interrupts?

- Can we cheaply emulate shared memory on message-passing HW?

Readings for Wed

- High-level programming models
- 4 tracks, let's divide up:
  - Task-parallel
  - Data-parallel
  - Pipeline-parallel
  - Implicit