CS 61C:
Great Ideas in Computer Architecture
OpenMP

Instructor:
Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa13

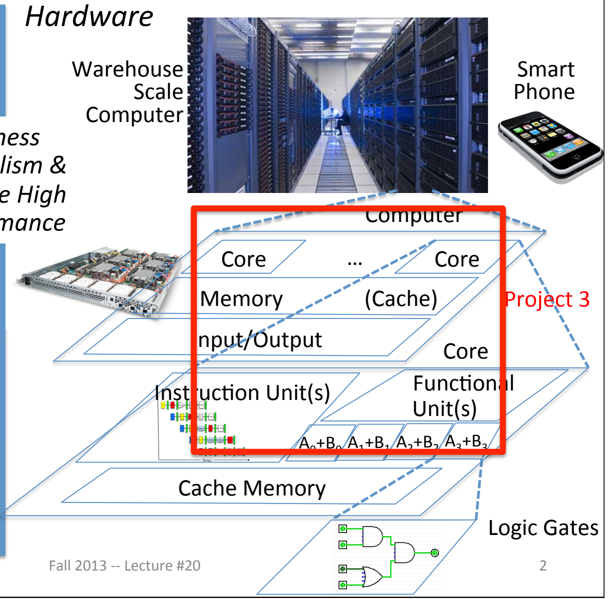
New-School Machine Structures (It's a bit more complicated!)

Software

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance

Hardware



Agenda

- Review
- openMP
- Administrivia
- PI and Matrix Multiplication Examples
- Scaling Experiments
- Technology Break
- False Sharing
- Synchronization
- And in Conclusion, …

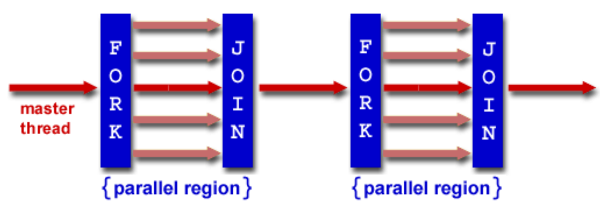
Agenda

- Review
- openMP
- Administrivia
- PI and Matrix Multiplication Examples
- Scaling Experiments
- Technology Break
- False Sharing
- Synchronization
- And in Conclusion, …

Review: OpenMP

- OpenMP is an API used for multi-threaded, shared memory parallelism
  - Compiler Directives (inserted into source code)
  - Runtime Library Routines (called from your code)
  - Environment Variables (set in your shell)
- Portable
- Standardized
- Easy to compile: `gcc –fopenmp name.c`

Review: Fork/Join Parallelism



- Start out executing the program with one master thread
- Master thread *forks* worker threads as enter parallel code
- Worker threads *join* (die or suspend) at end of parallel code

Image courtesy of http://www.llnl.gov/computing/tutorials/openMP/

Agenda

- Review
- openMP
- Administrivia
- PI and Matrix Multiplication Examples
- Scaling Experiments
- Technology Break
- False Sharing
- Synchronization
- And in Conclusion, …

The Parallel `for` Pragma

- Pragmas are a mechanism C provides for non-standard language extensions

```
#pragma omp parallel for
for (i=0; i<max; i++) zero[i] = 0;
```

- Master thread creates additional threads, each with a separate execution context
- Master thread becomes part of team of parallel threads inside parallel block

Controlling Number of Threads

- How many threads will OpenMP create?
  - Can set via clause in parallel pragma:

```
#pragma omp parallel for num_threads(NUM_THREADS)
```

  - or can set via explicit call to runtime function:

```
#include <omp.h> /* OpenMP header file. */
omp_set_num_threads(NUM_THREADS);
```

  - or via **NUM_THREADS** environment variable, usually set in your shell to the number of processors in computer running program
  - NUM_THREADS includes the master thread

What Kind of Threads?

- OpenMP threads are operating system threads
- OS multiplexes these onto available hardware threads
- Hopefully each assigned to a real hardware thread, so no OS-level time-multiplexing
- But other tasks on machine can also use those hardware threads!
- Be careful when timing results for project 3!

Invoking Parallel Threads

```
#include <omp.h>
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID);
}
```

Note: no for

- Each thread executes a copy of the code in the structured block
- OpenMP intrinsic to get Thread ID number: `omp_get_thread_num()`

Data Races and Synchronization

- Two memory accesses form a *data race* if from different threads access same location, at least one is a write, and they occur one after another
- If there is a data race, result of program varies depending on chance (which thread first?)
- Avoid data races by synchronizing writing and reading to get *deterministic* behavior
- Synchronization done by user-level routines that rely on hardware synchronization instructions
- (More on this later)

Controlling Sharing of Variables

- Variables declared outside parallel block are shared by default
- private(*x*) statement makes new private version of variable *x* for each thread

```
int i, temp, A[], B[];
#pragma omp parallel for private(temp)
for (i=0; i<N; i++) {
   temp = A[i]; A[i] = B[i]; B[i] = temp;
}
```

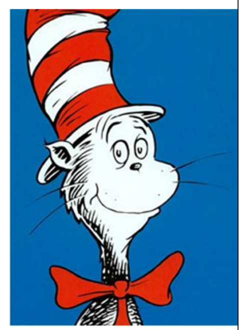
Note: for

Administrivia

- HW #5 posted
- Project 3 posted, 3-1 due Sunday@midnight
  - Image processing
  - Exploit what you are learning about cache blocking, SIMD instructions, and thread parallelism!
  - Who can achieve the fastest performance/highest speedup on the lab machines?

Katz ≠ Cats

- Cats:

Katz ≠ Cats

Katz (surname)

From Wikipedia, the free encyclopedia

Katz is a common German surname. It is also one of the oldest and most common Ashkenazi Jewish surnames.

Germans with the last name Katz may originate in the Rhine River region of Germany, where the Katz Castle is located. (The name of the castle does not derive from Katze, cat, but from Katzenelnbogen, going back to Latin Cattimelibocus, consisting of the ancient Germanic tribal name of the Chatti and Melibokus.)

Katzman, deriving from the German Katz, is a Slavic name meaning high priest or king. It is believed the Katzman surname originates from Germany and has roots from there as well.

As a Jewish surname, Katz is an abbreviation formed from the Hebrew initials of the term **Kohen Tzedeq** (Hebrew: כ״ץ), meaning "priest of justice"/"authentic priest" or **Kohen Tzadok** meaning the name-bearer is of patrilineal descent of the Kohanim sons of Zadok. It has been used since the seventeenth century, or perhaps somewhat earlier, as an epithet of the descendants of Aaron. The collocation is most likely derived from *Melchizedek* ("king of righteousness"), who is called *the priest ("kohen") of the most high God* (*Genesis* xiv. 18), or perhaps from Psalm cxxxii. 9: *Let thy priests be clothed with righteousness ("tzedeq").* The use of the abbreviated and Germanicized "Katz" likely coincided with the imposition of German names on Jews in Germany in the 18th or 19th centuries.

Agenda

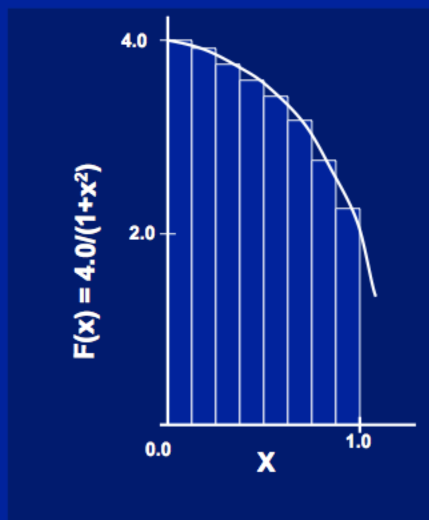
- Review
- openMP
- Administrivia
- PI and Matrix Multiplication Examples
- Scaling Experiments
- Technology Break
- False Sharing
- Synchronization
- And in Conclusion, …

π

3.
141592653589793238462643383279502
884197169399375105820974944592307
816406286208998628034825342117067
982148086513282306647093844609550
582231725359408128481117450284102
…

Calculating π

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Sequential Calculation of π in C

```
#include <stdio.h>/* Serial Code */
static long num_steps = 100000; double step;
int main (int argc; const char * argv[])
{    int i;       double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     for (i=1;i<= num_steps; i++){
       x = (i-0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
     }
     pi = sum/num_steps;
     printf ("pi = %6.12f\n", pi);
}
```

OpenMP Version (with bug)

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{    int i;     double  x, pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
#pragma omp parallel private (x)
{    int id = omp_get_thread_num();
     for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)
     {
         x = (i+0.5)*step;
         sum[id] += 4.0/(1.0+x*x);
     }
}
     for(i=0, pi=0.0; i<NUM_THREADS; i++)
       pi += sum[i] ;
   printf ("pi = %6.12f\n", pi  / num_steps);
}
```


OpenMP Version (with bug)

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{    int i;     double  x, pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
#pragma omp parallel private (x)
{    int id = omp_get_thread_num();
     for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)
     {
         x = (i+0.5)*step;
         sum[id] += 4.0/(1.0+x*x);
     }
}
     for(i=0, pi=0.0; i<NUM_THREADS; i++)
       pi += sum[i] ;
   printf ("pi = %6.12f\n", pi/num_steps);
}
```

Note: loop index variable i is shared between threads

OpenMP Reduction

- *Reduction*: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region: reduction(operation:var) where
  - *Operation*: operator to perform on the variables (var) at the end of the parallel region
  - *Var*: One or more variables on which to perform scalar reduction.

```
#pragma omp for reduction(+ : nSum)
  for (i = START ; i <= END ; ++i)
     nSum += i;
```


OpenMP Reduction Version

```
#include <omp.h>
#include <stdio.h>
/static long num_steps = 100000;
double step;
void main ()
{    int i;     double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
     for (i=1; i<= num_steps; i++){
      x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
     }
     pi = sum / num_steps;
  printf ("pi = %6.8f\n", pi);
}
```

Note: Don't have to declare for loop index variable i private, since that is default

OpenMP Timing

- omp_get_wtime – Elapsed wall-clock time

```
#include <omp.h> // to get function
double omp_get_wtime(void);
```

- Elapsed wall-clock time in seconds. The time is measured per thread, no guarantee can be made that two distinct threads measure the same time.
- Time is measured from some "time in the past". On POSIX-compliant systems the seconds since the Epoch (00:00:00 UTC, January 1, 1970) are returned.

Matrix Multiply in OpenMP

Note: Outer loop index i is private by default. Written explicitly here for clarity

```
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, i, j, k)
  for (i=0; i<Ndim; i++){
   for (j=0; j<Mdim; j++){
      tmp = 0.0;
      for(k=0;k<Pdim;k++){
         /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
         tmp += *(A+(i*Ndim+k)) *  *(B+(k*Pdim+j));
      }
      *(C+(i*Ndim+j)) = tmp;
   }
  }
run_time = omp_get_wtime() - start_time;
```

Note: Outer loop spread across N threads; inner loops inside a thread

Notes on Matrix Multiply Example

More performance optimizations available

- Higher compiler optimization (-O2) to reduce number of instructions executed
- Cache blocking to improve memory performance
- Using SIMD SSE3 Instructions to improve floating-point computation rate

Agenda

- Review
- openMP
- Administrivia
- PI and Matrix Multiplication Examples
- Scaling Experiments
- Technology Break
- False Sharing
- Synchronization
- And in Conclusion, …

32-Core System for Experiments

- Intel Nehalem Xeon 7550
- HW Multithreading: 2 Threads / core
- 8 cores / chip
- 4 chips / board

⇒ 64 Threads / system

- 2.00 GHz
- 256 KB L2 cache/ core
- 18 MB (!) shared L3 cache / chip

Experiments

- Compile and run at NUM_THREADS = 64
- Compile and run at NUM_THREADS = 64 with –O2
- Compile and run at NUM_THREADS = 32, 16, 8, … with –O2

Remember: Strong vs Weak Scaling

- Strong scaling: problem size fixed
- Weak scaling: problem size proportional to increase in number of processors
  - Speedup on multiprocessor while keeping problem size fixed is harder than speedup by increasing the size of the problem
  - But a natural use of a lot more performance is to solve a lot bigger problem

32 Core: Speed-up vs. Scale-up

Speed-up

Scale-up:
Fl. Pt. Ops = 2 x $Size^3$

Threads	Time	Speedup	Time	Size (Dim)	Fl. Ops x 10^9
1	13.75	1.00	13.75	1000	2.00
2			13.52	1240	3.81
4			13.79	1430	5.85
8			12.55	1600	8.19
16			13.61	2000	16.00
32			13.92	2500	31.25
64			13.83	2600	35.15

Memory Capacity = f($Size^2$), Compute = f($Size^3$)

32 Core: Speed-up vs. Scale-up

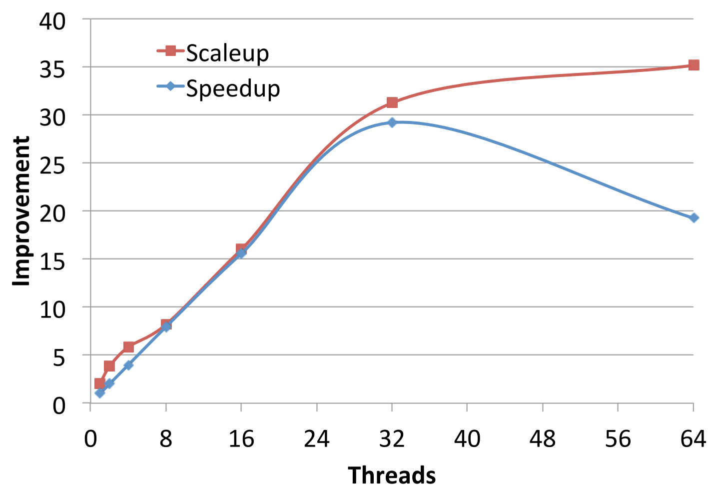
Speed-up

Scale-up:
Fl. Pt. Ops = 2 x $Size^3$

Threads	Time (secs)	Speedup	Time (secs)	Size (Dim)	Fl. Ops x 10^9
1	13.75	1.00	13.75	1000	2.00
2	6.88	2.00	13.52	1240	3.81
4	3.45	3.98	13.79	1430	5.85
8	1.73	7.94	12.55	1600	8.19
16	0.88	15.56	13.61	2000	16.00
32	0.47	29.20	13.92	2500	31.25
64	0.71	19.26	13.83	2600	35.15

Memory Capacity = f($Size^2$), Compute = f($Size^3$)

Strong vs. Weak Scaling



Peer Instruction: Why Multicore?

The switch in ~ 2004 from 1 processor per chip to multiple processors per chip happened because:

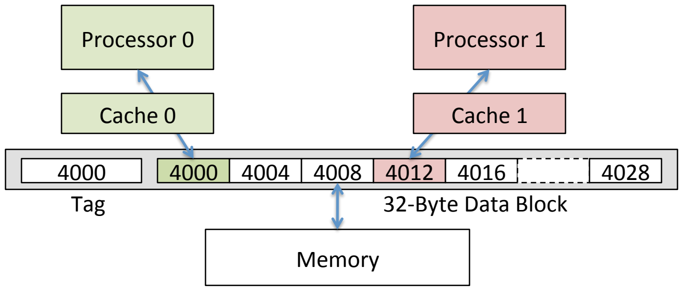
I. The "power wall" meant that no longer get speed via higher clock rates and higher power per chip
II. There was no other performance option but replacing 1 inefficient processor with multiple efficient processors
III. OpenMP was a breakthrough in ~2000 that made parallel programming easy

A)(orange) I only
B)(green) II only
C)(pink) I & II only
D)(yellow) I, II, & III

Agenda

- Review
- openMP
- Administrivia
- PI and Matrix Multiplication Examples
- Scaling Experiments
- Technology Break
- False Sharing
- Synchronization
- And in Conclusion, …

Cache Coherency Tracked by Block



- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?

Coherency Tracked by Cache Line

- Block ping-pongs between two caches even though processors are accessing disjoint variables
- Effect called *false sharing*
- How can you prevent it?

Fourth "C" of Cache Misses: *Coherence* Misses

- Misses caused by coherence traffic with other processor
- Also known as *communication* misses because represents data moving between processors working together on a parallel program
- For some parallel programs, coherence misses can dominate total misses

False Sharing in OpenMP

```
int i;      double  x, pi, sum[NUM_THREADS];
#pragma omp parallel private (i, x)
{   int id = omp_get_thread_num();
    for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREAD)
    {
       x = (i+0.5)*step;
       sum[id] += 4.0/(1.0+x*x);
    }
}
```

- What is problem?
- Sum[0] is 8 bytes in memory, Sum[1] is adjacent 8 bytes in memory => false sharing if block size > 8 bytes

Peer Instruction: No False Sharing

```
{   int i;    double  x, pi, sum[10000];
#pragma omp parallel private (i, x)
{   int id = omp_get_thread_num(),  fix = __________;
    for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS) {
        x = (i+0.5)*step;
        sum[id*fix] += 4.0/(1.0+x*x);
    }
```

- What is best value to set `fix` to prevent false sharing?

A)(orange) `omp_get_num_threads();`

B)(green) **Constant for number of blocks in cache**

C)(pink) **Constant for size of block in bytes**

D)(yellow) **Constant for size of blocks in doubles**

Agenda

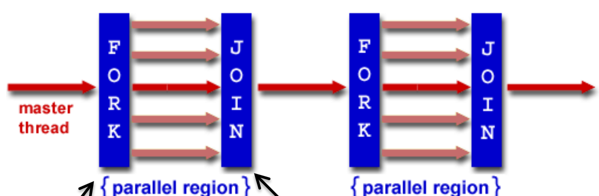
- Review
- openMP
- Administrivia
- PI and Matrix Multiplication Examples
- Scaling Experiments
- Technology Break
- False Sharing
- Synchronization
- And in Conclusion, …

Types of Synchronization

- Parallel threads run at varying speeds, need to synchronize their execution when accessing shared data.
- Two basic classes of synchronization:
  - Producer-Consumer
    - Consumer thread(s) wait(s) for producer thread(s) to produce needed data
    - Deterministic ordering. Consumer always runs after producer (unless there's a bug!)
  - Mutual Exclusion
    - Any thread can touch the data, but only one at a time.
    - Non-deterministic ordering. Multiple orders of execution are valid.

Simple OpenMP Parallel Sections

- OpenMP Fork and Join are examples of producer-consumer synchronization



Master doesn't fork worker threads until data is ready for them

At join, have to wait for all workers to finish at a "barrier" before starting following sequential master thread

Image courtesy of http://www.llnl.gov/computing/tutorials/openMP/

Barrier Synchronization

- Barrier waits for all threads to complete a parallel section. Very common in parallel processing.
- How does OpenMP implement this?

Barrier: First Attempt (pseudo-code)

```
int n_working = NUM_THREADS; /* Shared variable*/
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID); /* Do my chunk of work. */

  /* Barrier code. */
  n_working -= 1; /* I'm done */
  if (ID == 0) { /* Master */
    while (n_working != 0)
      ; /* master spins until everyone finished */
  } else {
     /* Put thread to sleep if not master */
  };
}
```

Flashcard quiz: Implementing Barrier Count decrement

- Thread #1

```
/* n_working -= 1 */
lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)
```

- Thread #2

```
/* n_working -=1 */
lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)
```

If initially **`n_working`** = 5, what are possible final values after both threads finish above code sequence?

- n_working = 3 only
- n_working = 3, or n_working = 4 only
- n_working = 3, 4, or 5 only
- Undefined

Decrement of Barrier Variable is Example of Mutual Exclusion

- Want each thread to *atomically* decrement the **`n_working`** variable
  - Atomic from Greek "Atomos" meaning indivisible!
- Ideally want:
  - Begin atomic section /*Only one thread at a time*/

```
lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)
```

  - End atomic section/*Allow another thread in */

New Hardware Instructions

For some common useful cases, some instruction sets have special instructions that atomically read-modify-write a memory location

Example:

```
fetch-and-add r_dest, (r_address), r_val implemented as:
  r_dest = Mem[r_address] //Return old value in register
  t = r_dest + r_val      // Updated value
  Mem[r_address] = t      //Increment value in memory
```

Simple common variant: **`test-and-set r_dest, (r_address)`**

Atomically reads old value of memory into r_dest, and puts 1 into memory location. Used to implement *locks*

Use locks for more general atomic sections

Atomic sections commonly called "critical sections"

Acquire(lock) /* Only one thread at a time in section. */
  /* Critical Section Code */
Release(lock) /* Allow other threads into section. */

- A lock is a variable in memory (one word)
- Hardware atomic instruction, e.g., test-and-set, checks and sets lock in memory

Implementing Barrier Count decrement with locks

```
/* Acquire lock */
spin:
testandset $t0, ($s1) /* $s1 has lock address */
bnez $t0, spin

lw $t0, ($s0)
addiu $t0, -1
sw $t0, ($s0)

/* Release lock */
sw $zero, ($s1)  /*Regular store releases lock*/
```


MIPS Atomic Instructions

- Splits atomic into two parts:
  - Load Linked **`LL rt, offset(base)`**
    - Regular load that "reserves" an address
  - Store Conditional **`SC rt, offset(base)`**
    - Store that only happens if no other hardware thread touched the reserved address
    - Success: rt=1 and memory updated
    - Failure: rt = 0 and memory unchanged
- Can implement test-and-set or fetch-and-add as short code sequence
- Reuses cache snooping hardware to check if other processors touch reserved memory location

ISA Synchronization Support

- All have some atomic Read-Modify-Write instruction
- Varies greatly – little agreement on "correct" way to do this
- No commercial ISA has direct support for producer-consumer synchronization
  - Use mutual exclusion plus software to get same effect (e.g., barrier in OpenMP)
- This area is still very much "work-in-progress" in computer architecture

OpenMP Critical Sections

```
#pragma omp parallel
{
  int ID = omp_get_thread_num();
  foo(ID); /* Do my chunk of work. */

#pragma omp critical
  { /* Only one thread at a time */
     /* shared_variable_updates */
  }
}
```

Agenda

- Review
- openMP
- Administrivia
- PI and Matrix Multiplication Examples
- Scaling Experiments
- Technology Break
- False Sharing
- Synchronization
- And in Conclusion, …

And, in Conclusion, …

- MatrixMultiply speedup versus scaleup
  - Strong versus weak scaling
- Synchronization:
  - Producer-consumer versus mutual-exclusion
- Hardware provides some atomic instructions
  - Software builds up other synchronization using these