CS 61C:
Great Ideas in Computer Architecture
OpenMP, Part I

Instructor:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

Review

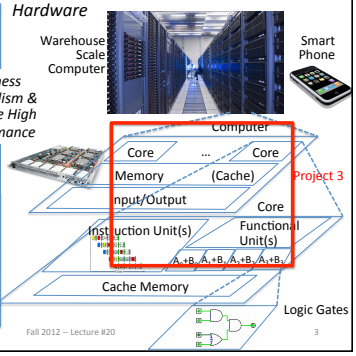
- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multiprocessor/Multicore uses Shared Memory
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!
- Multithreading increases utilization, Multicore more processors (MIMD)

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

Software *Hardware*

- 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



100s of (mostly dead) Parallel Programming Languages

ActorScript	Concurrent Pascal	JoCaml	Orc
Ada	Concurrent ML	Join	Oz
Afnix	Concurrent Haskell	Java	Pict
Alef	Curry	Joule	Reia
Alice	CUDA	Joyce	SALSA
APL	E	LabVIEW	Scala
Axum	Eiffel	Limbo	SISAL
Chapel	Erlang	Linda	SR
Cilk	Fortan 90	MultiLisp	Stackless Python
Clean	Go	Modula-3	SuperPascal
Clojure	Io	Occam	VHDL
Concurrent C	Janus	occam-π	XC

OpenMP

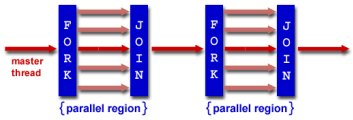
- OpenMP is an API used for multi-threaded, shared memory parallelism
  - Compiler Directives (inserted into source code)
  - Runtime Library Routines (called from your code)
  - Environment Variables (set in your shell)
- Portable
- Standardized
- Easy to compile: `cc –fopenmp name.c`

Simple Parallelization

```
for (i=0; i<max; i++) zero[i] = 0;
```

- For loop must have canonical shape for OpenMP to parallelize it
  - Necessary for run-time system to determine loop iterations
- No premature exits from the loop allowed
  - i.e., No break, return, exit, goto statements

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/

OpenMP Extends C with Pragmas

- Pragmas are a mechanism C provides for non-standard language extensions
  - **`#pragma description`**
- Commonly implemented pragmas: structure packing, symbol aliasing, floating-point exception modes
- Good mechanism for OpenMP because compilers that don't recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas

The parallel `for` pragma

```
#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`** an 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 will multiplex requested OpenMP threads onto available hardware threads.
- Hopefully each get a real hardware thread to run on, so no OS-level time-multiplexing.
- But other tasks on machine can also use 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);
}
```

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

Data Races and Synchronization

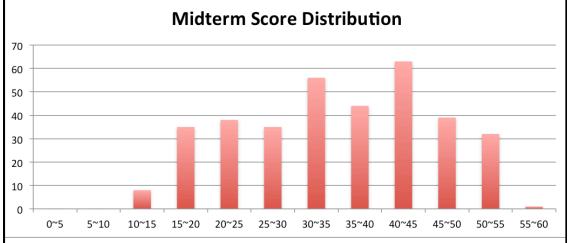
- 2 memory accesses form a *data race* if from different threads to same location, and at least one is a write, and they occur one after another
- If there is a data race, result of program can vary depending on chance (which thread first?)
- Avoid data races by synchronizing writing and reading to get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


Administrivia



Only 2/3rd of grades included above

Regrade Request Policy

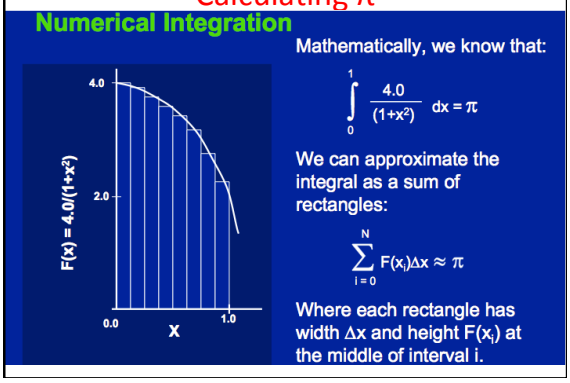
- NO REQUESTS ACCEPTED UNTIL LECTURE WED OCTOBER 17, i.e., we'll simply delete any that come before
- Must attend discussion section to learn solutions and grading process – TA signoff needed for regrade request!
- Regrade requests must be accompanied by written request explaining rationale for regrade.
- Modifying your copy of exam punishable by F and letter in your University record
- We reserve right to regrade whole exam

π

3.
14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410...

- Pi Day is 3-14 (started at SF Exploratorium)

Calculating π



Sequential Calculation of π in C

```
#include <stdio.h>/* Serial Code */
static long num_steps = 100000; double step;
void main ()
{    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);
}
```


Experiment

- Run with NUM_THREADS = 1 multiple times
- Run with NUM_THREADS = 2 multiple times
- What happens?

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 Version 2 (with bug)

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


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

And in Conclusion, …

- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, critical sections, …
  - ≈ C: small so easy to learn, but not very high level and its easy to get into trouble