UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Professor
Bora Nikolić

Thread-Level Parallelism II

Berkeley
UNIVERSITY OF CALIFORNIA

cs61c.org

Parallel Programming Languages

Languages Supporting Parallel Programming

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

Which one to pick?

Why So Many Parallel Programming Languages?

- Why "intrinsics"?
  - TO Intel: fix your #()&$! compiler, thanks…
- It's happening ... but
  - SIMD features are continually added to compilers (Intel, gcc)
  - Intense area of research
  - Research progress:
    - 20+ years to translate C into good (fast!) assembly
    - How long to translate C into good (fast!) parallel code?
      - General problem is very hard to solve
      - Present state: specialized solutions for specific cases
      - Your opportunity to become famous!

Garcia, Nikolić

Parallel Programming Languages

- Number of choices is indication of
  - No universal solution
    - Needs are very problem specific
  - E.g.,
    - Scientific computing/machine learning (matrix multiply)
    - Webserver: handle many unrelated requests simultaneously
    - Input / output: it's all happening simultaneously!
- Specialized languages for different tasks
  - Some are easier to use (for some problems)
  - None is particularly "easy" to use
- 61C
  - Parallel language examples for high-performance computing
  - OpenMP

Garcia, Nikolić

OpenMP

Parallel Loops

- Serial execution:

```
for (int i=0; i<100; i++) {
        …
}
```

- Parallel Execution:

`for (int i=0; i<25; i++) { … }`	`for (int i=25; i<50; i++) { … }`	`for (int i=50; i<75; i++) { … }`	`for (int i=75; i<100; i++) { … }`

Garcia, Nikolić

Parallel `for` in OpenMP

```
#include <omp.h>

#pragma omp parallel for
for (int i=0; i<100; i++) {
    …
}
```


Garcia, Nikolić

OpenMP Example

```c
/* clang -Xpreprocessor -fopenmp -lomp -o for for.c */

#include <stdio.h>
#include <omp.h>
int main()
{
    omp_set_num_threads(4);
    int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int N = sizeof(a)/sizeof(int);

    #pragma omp parallel for
    for (int i=0; i<N; i++) {
        printf("thread %d, i = %2d\n",
            omp_get_thread_num(), i);
        a[i] = a[i] + 10 * omp_get_thread_num();
    }

    for (int i=0; i<N; i++) printf("%02d ", a[i]);
    printf("\n");
}
```

```
$ gcc-5 -fopenmp for.c;./a.out

% gcc -Xpreprocessor -fopenmp -
lomp -o for for.c; ./for
thread 0, i =  0
thread 1, i =  3
thread 2, i =  6
thread 3, i =  8
thread 0, i =  1
thread 1, i =  4
thread 2, i =  7
thread 3, i =  9
thread 0, i =  2
thread 1, i =  5
00 01 02 13 14 15 26 27 38 39
```

The call to find the maximum number of threads that are available to do work is **`omp_get_max_threads()`** (from **`omp.h`**).

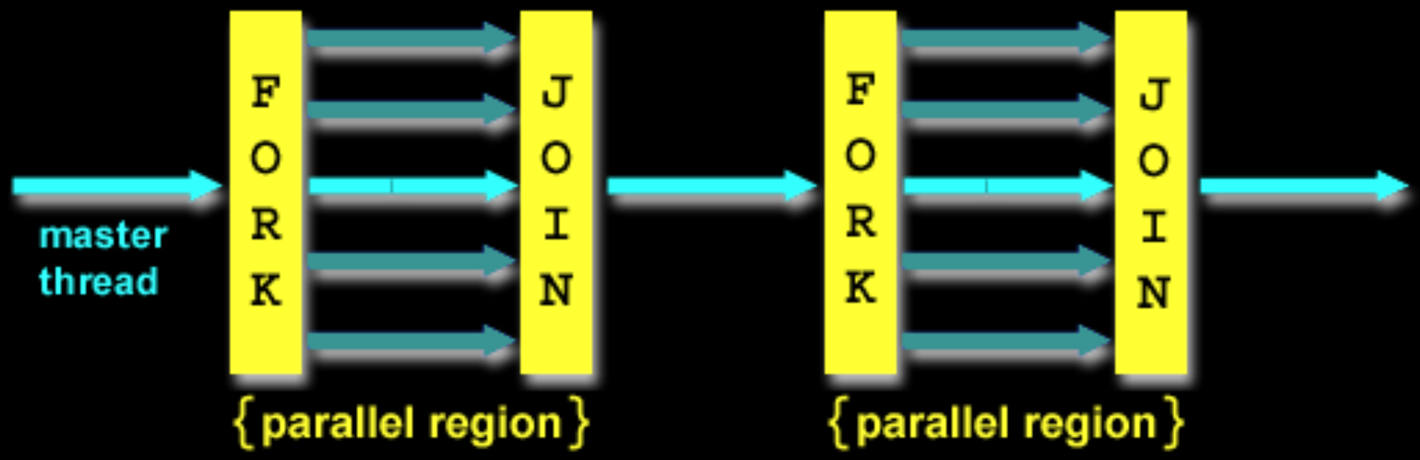
OpenMP

- C extension: no new language to learn
- Multi-threaded, shared-memory parallelism
  - Compiler Directives, `#pragma`
  - Runtime Library Routines, `#include <omp.h>`
- `#pragma`
  - Ignored by compilers unaware of OpenMP
  - Same source for multiple architectures
    - E.g., same program for 1 & 16 cores
- Only works with shared memory

Garcia, Nikolić

OpenMP Programming Model

- Fork - Join Model:



- OpenMP programs begin as single process *(main thread)*
  - Sequential execution
- When parallel region is encountered
  - Master thread "forks" into team of parallel threads
  - Executed simultaneously
  - At end of parallel region, parallel threads "join", leaving only master thread
- Process repeats for each parallel region
  - Amdahl's Law?

What Kind of Threads?

- OpenMP threads are operating system (software) threads
- OS will multiplex requested OpenMP threads onto available hardware threads
- Hopefully each gets a real hardware thread to run on, so no OS-level time-multiplexing
- But other tasks on machine compete for hardware threads!
- Be "careful" (?) when timing results for Projects!
  - 5AM?
  - Job queue?

☑

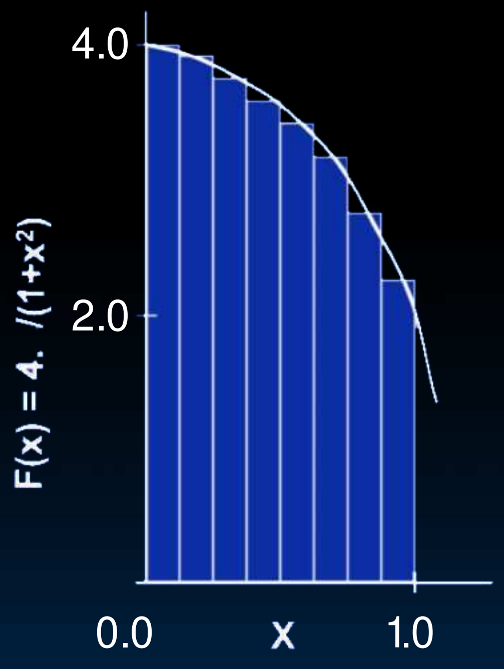
Computing π

Example 2: Computing π

```
In[1]:= Integrate[ 4*Sqrt[1-x^2] , {x,0,1}]  ← Tested using Mathematica
Out[1]= Pi

In[2]:= Integrate[ (4/(1+x^2)) , {x,0,1}]
Out[2]= Pi
```

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.

http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Sequential π = 3.1415926535897932384626433832795028841971693993751...

```c
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5)*step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

`pi = 3.142425985001`

- Resembles π, but not very accurate
- Let's increase **`num_steps`** and parallelize

Parallelize (1) …

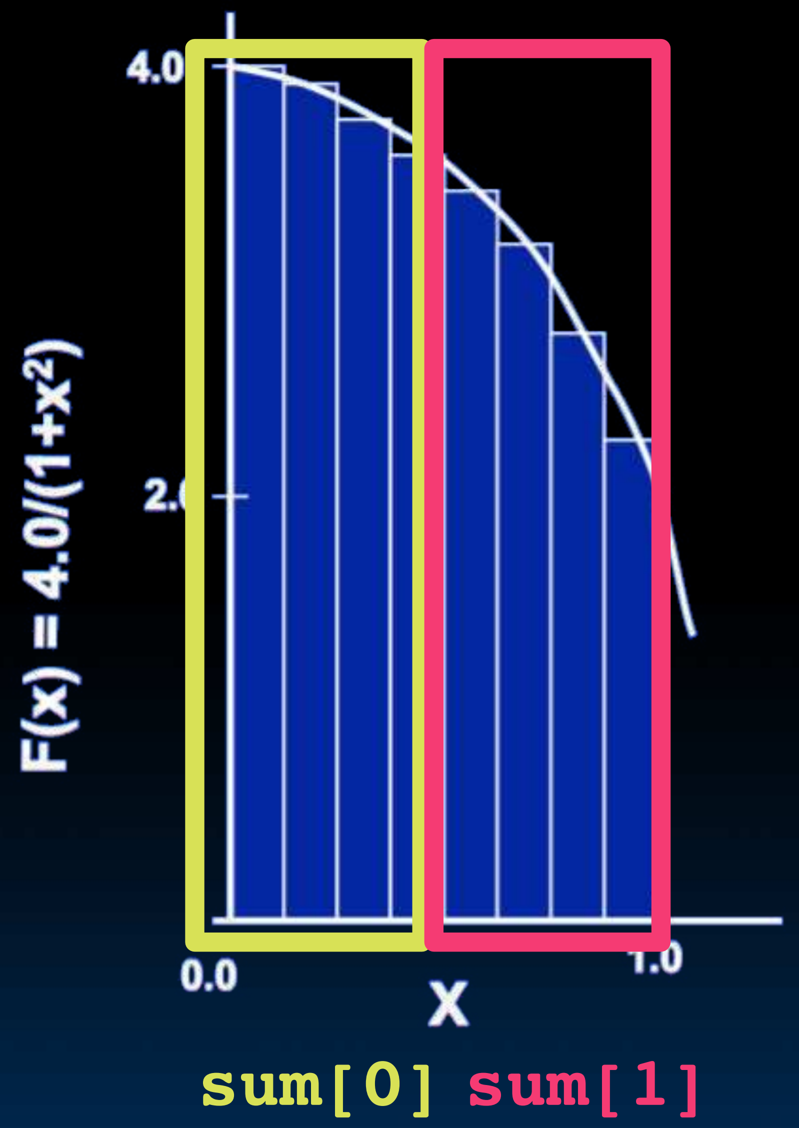
```c
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
#pragma parallel for
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

- Problem: each thread needs access to the shared variable **sum**
- Code runs sequentially …

Garcia, Nikolić

Parallelize (2) …



1. Compute
   `sum[0]` and `sum[1]`
   in parallel

2. Compute
   `sum = sum[0]+sum[1]`
   sequentially

Garcia, Nikolić

Parallel π … Trial Run

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

```
i =  1,  id =  1
i =  0,  id =  0
i =  2,  id =  2
i =  3,  id =  3
i =  5,  id =  1
i =  4,  id =  0
i =  6,  id =  2
i =  7,  id =  3
i =  9,  id =  1
i =  8,  id =  0
pi = 3.142425985001
```

Scale up: `num_steps = 10`^{6}

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 1000000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            // printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

`pi = 3.141592653590`

You verify how many digits are correct …

Can We Parallelize Computing sum?

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Always looking for ways to beat Amdahl's Law …

Summation inside parallel section

- Insignificant speedup in this example, but …
- pi = 3.138450662641
- Wrong! And value changes between runs?!
- What's going on?

What's Going On?

```c
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

- Operation is really
  `pi = pi + sum[id]`
- What if >1 threads reads current (same) value of `pi`, computes the sum, stores the result back to `pi`?
- Each processor reads same intermediate value of `pi`!
- Result depends on who gets there when
  - A "race" → result is <u>not deterministic</u>

☑

Synchronization

Synchronization

- Problem:
  - Limit access to shared resource to 1 actor at a time
  - E.g. only 1 person permitted to edit a file at a time
    - otherwise changes by several people get all mixed up

- Solution:

- Take turns:
  - Only one person get's the microphone & talks at a time
  - Also good practice for classrooms, btw …

Locks

- Computers use locks to control access to shared resources
  - Serves purpose of microphone in example
  - Also referred to as "semaphore"

- Usually implemented with a variable
  - `int lock;`
    - 0 for unlocked
    - 1 for locked

Garcia, Nikolić

Synchronization with Locks

```
// wait for lock released
while (lock != 0) ;
// lock == 0 now (unlocked)

// set lock
lock = 1;

     // access shared resource ...
     // e.g. pi
     // sequential execution! (Amdahl ...)

// release lock
lock = 0;
```


Garcia, Nikolić

Lock Synchronization

Thread 1	Thread 2
`while (lock != 0) ;`	
	`while (lock != 0) ;`
`lock = 1;`	
`// critical section`	
	`lock = 1;`
	`// critical section`
`lock = 0;`	
	`lock = 0;`

- Thread 2 finds lock not set, before thread 1 sets it
- Both threads believe they got and set the lock!

Try as you like, this problem has no solution, not even at the assembly level.
Unless we introduce new instructions, that is! (next lecture)

And, in Conclusion, …

- OpenMP as simple parallel extension to C
  - Threads level programming with **`parallel for`** pragma
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble
- Race conditions – result of program depends on chance (bad)
  - Need assembly-level instructions to help with lock synchronization
  - …next time