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)

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`

100s of (mostly dead) Parallel Programming Languages

<table>
<thead>
<tr>
<th>Software</th>
<th>Hardware</th>
<th>Smart Phone Warehouse Scale Computer</th>
<th>New-School Machine Structures (It's a bit more complicated!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActorScript</td>
<td>Concurrent Pascal</td>
<td>JoCaml</td>
<td>Ada</td>
</tr>
<tr>
<td>Ada</td>
<td>Concurrent ML</td>
<td>Join</td>
<td>Afhix</td>
</tr>
<tr>
<td>Afnix</td>
<td>Concurrent Haskell</td>
<td>Java</td>
<td>Pict</td>
</tr>
<tr>
<td>Alf</td>
<td>Curry</td>
<td>Joule</td>
<td>Reia</td>
</tr>
<tr>
<td>Alice</td>
<td>CUDA</td>
<td>Joyce</td>
<td>Salsa</td>
</tr>
<tr>
<td>APL</td>
<td>E</td>
<td>LabVIEW</td>
<td>Scala</td>
</tr>
<tr>
<td>Axum</td>
<td>Eiffel</td>
<td>Limbo</td>
<td>SISAL</td>
</tr>
<tr>
<td>Chapel</td>
<td>Erlang</td>
<td>Linda</td>
<td>SR</td>
</tr>
<tr>
<td>Clean</td>
<td>Go</td>
<td>Modula-3</td>
<td>SuperPascal</td>
</tr>
<tr>
<td>Clojure</td>
<td>Io</td>
<td>Occam</td>
<td>VHDL</td>
</tr>
<tr>
<td>Concurrent C</td>
<td>Janus</td>
<td>occam-n</td>
<td>XC</td>
</tr>
</tbody>
</table>

Simple Parallelization

```c
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

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

```c
#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

```c
#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:
  - #pragma omp parallel for num_threads(NUM_THREADS)

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

```c
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

**Midterm Score Distribution**

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

Calculating π

3. 
141592653589793238462643383279502
84197169399375105820974944592307
816406286208988628034825342117067
982148086513282306647093844609550
582231725359408128481117450284102 ...
- Pi Day is 3-14 (started at SF Exploratorium)
Sequential Calculation of π in C

```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)

```c
#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=id+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 2 (with bug)

```c
#include <omp.h>

static long num_steps = 100000; double step;
#define NUM_THREADS 2

void main ()
{
    int i; double x, 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=id+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.

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

```c
#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