You Are Here!

- 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 functioning in parallel at same time

Review: Synchronization in MIPS

- Load linked: \texttt{ll \textit{rt}, offset(rs)}
- Store conditional: \texttt{sc \textit{rt}, offset(rs)}
  - Succeeds if location not changed since the \texttt{ll}
  - Returns 1 in rt (clobbers register value being stored)
  - Fails if location has changed
  - Returns 0 in rt (clobbers register value being stored)
- Example: atomic swap (to test/set lock variable)

Exchange contents of reg and mem: \texttt{:$s4 \leftrightarrow ($s1)}

\begin{verbatim}
try: add $t0,$zero,$s4 ;copy exchange value
 ll $t1,0($s1)    ;load linked
 sc $t0,0($s1)    ;store conditional
 beq $t0,$zero,try ;branch store fails
 add $s4,$zero,$t1 ;put load value in $s4
\end{verbatim}

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Examples
- Technology Break
- Common Pitfalls
- Summary
Machines in 61C Lab

```bash
/usr/sbin/sysctl -a | grep hw\.
```

- `hw.model = MacPro4,1`
- `hw.cachesize = 64`
- `hw.physicalcpu: 8`
- `hw.logicalcpu: 16`
- `hw.physicalcpu: 2,260,000,000`
- `hw.physmem = 2,147,483,648`

**Try up to 16 threads to see if performance gain even though only 8 cores**

Randy’s Laptop

```bash
hw.model = MacBookAir3,1
```

- `hw.cachesize = 64`
- `hw.physicalcpu: 2`
- `hw.logicalcpu: 2`
- `hw.cachesize = 3,145,728`
- `hw.physicalcpu: 1,600,000,000`
- `hw.physmem = 2,147,483,648`

**No l3 cache**

- Dual core
- One hw context per core

What is OpenMP?

- API used for multi-threaded, shared memory parallelism
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable
- Standardized

Shared Memory Model with Explicit Thread-based Parallelism

- Shared memory process consists of multiple threads, explicit programming model with full programmer control over parallelization
- Pros:
  - Takes advantage of shared memory, programmer need not worry (that much) about data placement
  - Programming model is “serial-like” and thus conceptually simpler than alternatives (e.g., message passing/MPI)
  - Compiler directives are generally simple and easy to use
  - Legacy serial code does not need to be rewritten
- Cons:
  - Codes can only be run in shared memory environments!
  - Compiler must support OpenMP (e.g., gcc 4.2)

OpenMP Uses the C Extension Pragmas Mechanism

- Pragmas are a preprocessor mechanism that C provides for language extensions
- Many different uses: structure packing, symbol aliasing, floating point exception modes, ...
- Good for OpenMP because compilers that don’t recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas
OpenMP Programming Model

• Fork-Join Model:
  - Fork: the master thread then creates a team of parallel threads
  - Join: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

OpenMP Directives

Building Block: C for loop

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

• Break for loop into chunks, and allocate each to a separate thread
  - E.g., if max = 100, with two threads, assign 0-49 to thread 0, 50-99 to thread 1

OpenMP: 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
• All variables declared outside for loop are shared by default, except for loop index which is private per thread (Why?)
• Implicit synchronization at end of for loop
• Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, ..., (max/n)-1;
  - Thread 1 gets max/n, max/n+1, ..., 2*(max/n)-1
  - Why?

Thread Creation

• How many threads will OpenMP create?
• Defined by OMP_NUM_THREADS environment variable (or in code procedure call)
• Set this variable to the maximum number of threads you want OpenMP to use
• Usually equals the number of cores in the underlying HW on which the program is run

OMP_NUM_THREADS

• Shell command to set number threads:
  ```sh
echo OMP_NUM_THREADS=x
```
• Shell command check number threads:
  ```sh
echo SOMP_NUM_THREADS
```
• OpenMP intrinsic to set number of threads:
  ```c
  omp_num_threads(x);
  ```
• OpenMP intrinsic to get number of threads:
  ```c
  num_th = omp_get_num_threads();
  ```
• OpenMP intrinsic to get Thread ID number:
  ```c
  th_ID = omp_get_thread_num();
  ```
Parallel Threads and Scope

- Each thread executes a copy of the code within the structured block
  
  ```
  #pragma omp parallel
  {
      ID = omp_get_thread_num();
      foo(ID);
  }
  ```

- OpenMP default is shared variables
  - To make private, need to declare with pragma
    
    ```
    #pragma omp parallel private (x)
    ```

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Examples
- Technology Break
- Common Pitfalls
- Summary

Administrivia

- Regrade Policy
  - Rubric on-line
  - Questions covered in Discussion Section this week
  - Written appeal process
    - Explain rationale for regrade request
    - Attach rationale to exam
    - Submit to your TA in this week’s laboratory

Administrivia

- Next Lab and Project
  - Lab #9: Thread Level Parallelism, this week
  - HW #5: due March 27, Make and Threads
  - Project #3: Matrix Multiply Performance Improvement
    - Work in groups of two!
    - Stick around after class for shotgun marriage
    - Part 1: March 27 (end of Spring Break)
    - Part 2: April 3

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Examples
- Technology Break
- Common Pitfalls
- Summary
OpenMP Examples

- Hello World
- OMP Get Environment (hidden slides)
- For Loop Workshare
- Section Workshare
- Sequential and Parallel Numerical Calculation of Pi

Hello World in OpenMP

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

int main(void) {
    int nthreads,
        tid;

    /* Fork team of threads with each having a private tid variable */
    #pragma omp parallel private(tid)
    {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
    /* All threads join master thread and terminate */
}
```

localhost:OpenMP randykatz$ ./omp_hello
Hello World from thread = 0
Hello World from thread = 1
Number of threads = 2

Hello World in OpenMP

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

int main(int argc, char *argv[]) {
    int nthreads,
        tid,
        procs,
        maxt,
        inpar,
        dynamic,
        nested;

    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num(); /* Obtain thread number */

        /* Only master thread does this */
        if (tid == 0) {
            printf("Thread %d getting environment info...\n", tid);
            /* Get environment information */
            procs = omp_get_num_procs();
            nthreads = omp_get_num_threads();
            maxt = omp_get_max_threads();
            inpar = omp_in_parallel();
            dynamic = omp_get_dynamic();
            nested = omp_get_nested();

            printf("Number of processors = %d\n", procs);
            printf("Number of threads = %d\n", nthreads);
            printf("Max threads = %d\n", maxt);
            printf("In parallel? = %d\n", inpar);
            printf("Dynamic threads enabled? = %d\n", dynamic);
            printf("Nested parallelism supported? = %d\n", nested);
        }
    }
}
```

Get Environment Information in OpenMP

```c
/* Print environment information */
printf("Number of processors = %d\n", procs);
printf("Number of threads = %d\n", nthreads);
printf("Max threads = %d\n", maxt);
printf("In parallel? = %d\n", inpar);
printf("Dynamic threads enabled? = %d\n", dynamic);
printf("Nested parallelism supported? = %d\n", nested);
}
/* Parallel Done */
```

Get Environment Information in OpenMP

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

int main(int argc, char *argv[]) {
    int nthreads,
        tid,
        procs,
        maxt,
        inpar,
        dynamic,
        nested;

    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num(); /* Obtain thread number */
        if (tid == 0) {
            /* Only master thread does this */
            printf("Thread %d getting environment info...\n", tid);
            /* Get environment information */
            procs = omp_get_num_procs();
            nthreads = omp_get_num_threads();
            maxt = omp_get_max_threads();
            inpar = omp_in_parallel();
            dynamic = omp_get_dynamic();
            nested = omp_get_nested();

            printf("Number of processors = %d\n", procs);
            printf("Number of threads = %d\n", nthreads);
            printf("Max threads = %d\n", maxt);
            printf("In parallel? = %d\n", inpar);
            printf("Dynamic threads enabled? = %d\n", dynamic);
            printf("Nested parallelism supported? = %d\n", nested);
        }
    }
}
```

Get Environment Information in OpenMP

```c
/* Print environment information */
printf("Thread 0 getting environment info...
Number of processors = %d\n", procs);
printf("Number of threads = %d\n", nthreads);
printf("Max threads = %d\n", maxt);
printf("In parallel? = %d\n", inpar);
printf("Dynamic threads enabled? = %d\n", dynamic);
printf("Nested parallelism supported? = %d\n", nested);
```

Executes two threads
Prints information from thread 0 only
At this point just one thread available, hence max threads = 1
Workshare for loop Example #1 in OpenMP

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

#define N 100

int main(int argc, char *argv[])
{
    int *a[N], *b[N], *c[N], *d[N];
    float *x[N], *y[N], *z[N];

    /\ Some initializations /*
    for (i=0; i<N; i++)
    {  
        a[i] = b[i] = c[i] = d[i] = 0.0;
    }

    /* Start of parallel section */
    for (i=0; i<N; i++)
    {
        x[i] = y[i] = z[i] = 2.0;
        c[i] = a[i] * b[i];

        #pragma omp task
        {
            #pragma omp single
            {
                printf("Thread %d doing section 1\n",tid);
                for (j=0; j<i; j++)
                    c[i] += a[j] * b[j];
                printf("Thread %d doing section 2\n",tid);
            }
        }
        #pragma omp taskwait
    }
    /\ End of parallel section */

    return 0;
}

Workshare for loop Example #1 in OpenMP

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

#define N 50

int main (int argc, char *argv[])
{
    int *a[N], *b[N], *c[N], *d[N];
    float *x[N], *y[N], *z[N];

    /\ Some initializations /*
    for (i=0; i<N; i++)
    {  
        a[i] = b[i] = c[i] = d[i] = 0.0;
    }

    /* Start of parallel section */
    for (i=0; i<N; i++)
    {
        x[i] = y[i] = z[i] = 2.0;
        c[i] = a[i] * b[i];

        #pragma omp task
        {
            #pragma omp single
            {
                printf("Thread %d doing section 1\n",tid);
                for (j=0; j<i; j++)
                    c[i] += a[j] * b[j];
                printf("Thread %d doing section 2\n",tid);
            }
        }
        #pragma omp taskwait
    }
    /\ End of parallel section */

    return 0;
}
Workshare sections Example #2 in OpenMP

```
#pragma omp section
{
    printf("Thread %d doing section %d\n", tid, i);
    d[i] = a[i] * b[i];
    printf("Thread %d: d[%d] = %f\n", tid, i, d[i]);
}
} /* end of sections */
```

Workshare sections Example #2 in OpenMP

```
printf("Thread %d: c[%d] = %f\n", tid, i, c[i]);
}
} /* end of parallel section */
```

Calculating $\pi$ using Numerical Integration

$$\pi = \int_0^1 \frac{4}{1 + x^2} \, dx$$

Riemann Sum Approximation:

$$\int_a^b f(x) \, dx \approx \left[ \sum_{i=1}^{n} f(x_i) \Delta x \right]$$

where $\Delta x = \frac{b-a}{n}$ and $x_i$ is the midpoint of the $i$th subinterval, $[x_{i-1}, x_i]$. $f(x)$ is the function $\frac{4}{1 + x^2}$.

Sequential Calculation of $\pi$ in C

```
#include <stdio.h>
/* Serial Code */
static long num_steps = 100000;
double step;
int main(int argc, char *argv[]) {
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0; i < num_steps; i++) {
        x = (i+0.5)*step;
        pi = step/num_steps;
        printf("\pi = \%f\n", pi);
        }
    return 0;
}
```

http://www.youtube.com/watch?v=H2bKjz-bjw
OpenMP Version #1

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

OpenMP Version #1 (with bug)

```c
pi = 2.187926636214  pi = 3.141592653598
pi = 2.093066397959  pi = 1.999849495043
pi = 2.129235450053  pi = 2.348751552706
pi = 2.19973153499    pi = 3.15192653598
pi = 2.128710582772   pi = 2.42597425566
pi = 2.120057931672   pi = 3.141592653598
pi = 1.97973212760    pi = 2.024963879529
pi = 2.14406923694    pi = 2.357897349032
pi = 2.141592653598   pi = 2.325698828349
pi = 2.273284475646   pi = 1.825693174923
```

OpenMP Critical Section

```c
#include <omp.h>
int main(void) {
    int x;
    x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x + 1;
    }
    /* end of parallel section */
}
```

OpenMP Version #2

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

OpenMP Version #2 (with bug)

```c
pi = 2.19066170635  pi = 3.141592653598
pi = 2.094493774172  pi = 2.44654596856
pi = 2.94118318377  pi = 2.292318972127
pi = 3.07688916821  pi = 3.430000515865
pi = 3.146809441280  pi = 3.6502903478614
pi = 3.146534129677  pi = 1.988309636488
pi = 3.141592653598  pi = 3.141592653598
pi = 2.880126980443  pi = 2.516382108613
pi = 2.021271161080  pi = 3.141592653598
pi = 2.94646689461    pi = 2.331398287079
pi = 2.769957838388   pi = 2.245873892167
pi = 2.373218039482   pi = 1.928775146184
```

OpenMP Reduction

- **Reduction**: specifies that one 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 Version #3

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

static long num_steps = 100000;
double step;
int main (int argc, char *argv[])
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    
    #pragma omp parallel for private(x)
    for (i = 0; i < num_steps; i++) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    
    pi = sum / num_steps;
    
    printf("pi = \%6.8f\n", pi);
}
```

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

OpenMP Timing

- `omp_get_wtime` - Elapsed wall clock time
  ```c
  double omp_get_wtime(void);
  ```
  
  #include <omp.h> // to get function

  Elapsed wall clock time in seconds. 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.

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Examples
- Technology Break
- Common Pitfalls
- Summary

OpenMP Version #4

```c
#include <omp.h>

static long num_steps = 100000; double step;

int main (int argc, char *argv[])
{
    int i, id; double x, pi, sum; 
    sum = 0.0; 
    step = 1.0/(double) num_steps;
    
    #pragma omp parallel
    
        #pragma omp for 
        for (i = 0; i < num_steps; i++) 
            { 
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
            }
    
    printf("pi = \%6.12f\n", pi*sum);
}
```
**OpenMP Version #4 (with bug)**

```
pi = 1.2653703708888
pi = 1.9617540079590
pi = 1.7026309071404
pi = 3.1135076595006
pi = 3.0762666475686
pi = 3.0338581534000
pi = 3.1415926535980
pi = 3.1415926535980
```

```c
sum0 += 4.0/(1.0+x*step);  
```

```
3/14/11
```
Notes on Matrix Multiply Example

More performance optimizations available
- Higher compiler optimization (-O2, -O3) to reduce number of instructions executed
- Cache blocking to improve memory performance
- Using SIMD SSE3 Instructions to raise floating point computation rate

OpenMP Pitfall #1: Data Dependencies

- Consider the following code:
  
  ```plaintext
  a[0] = 1;
  for (i=1; i<5; i++)
    a[i] = i + a[i-1];
  ```

  - There are dependencies between loop iterations
  - Sections of loops split between threads will not necessarily execute in order
  - Out of order loop execution will result in undefined behavior

OpenMP Pitfall #2: Avoiding Dependencies by Using Private Variables

- Consider the following loop:
  ```plaintext
  for (i=0; i<n; i++)
    temp = 2.0*a[i];
    a[i] = temp;
    b[i] = c[i]/temp;
  ```

  - Threads share common address space: will be modifying temp simultaneously; solution:
    ```plaintext
    #pragma omp parallel for private(temp)
    ```
  
  - Compiler can generate highly efficient code for reduction

OpenMP Pitfall #3: Parallel Overhead

- Spawning and releasing threads results in significant overhead
- Therefore, you want to make your parallel regions as large as possible
  - Parallelize over the largest loop that you can (even though it will involve more work to declare all of the private variables and eliminate dependencies)
  - Coarse granularity is your friend!

And in Conclusion, ...

- 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!
- Data races lead to subtle parallel bugs
- Synchronization via atomic operations:
  - MIPS does it with Load Linked + Store Conditional
- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, critical sections, ...