A Programmer’s View of OpenMP

• What is OpenMP?
  • Open specification for Multi-Processing
  • “Standard” API for defining multi-threaded shared-memory programs
  • openmp.org – Talks, examples, forums, etc.

• OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  • Exact behavior depends on OpenMP implementation!
  • Requires compiler support (C or Fortran)

• OpenMP will:
  • Allow a programmer to separate a program into serial regions and parallel regions, rather than T concurrently-executing threads.
  • Hide stack management
  • Provide synchronization constructs

• OpenMP will not:
  • Parallelize automatically
  • Guarantee speedup
  • Provide freedom from data races
Motivation – OpenMP

int main() {

    // Do this part in parallel

    printf("Hello, World!\n");

    return 0;
}

Motivation – OpenMP

int main() {

    omp_set_num_threads(4);

    // Do this part in parallel
    #pragma omp parallel
    {
        printf( "Hello, World!\n" );
    }

    return 0;
}
OpenMP parallel region construct

- Block of code to be executed by multiple threads in parallel
- Each thread executes the **same code redundantly (SPMD)**
  - Work within work-sharing constructs is distributed among the threads in a team
- Example with C/C++ syntax

```c
#pragma omp parallel [ clause [ clause ]... ] new-line structured-block
```

- clause can include the following:
  - `private (list)`
  - `shared (list)`
- Example: OpenMP default is `shared` variables
To make private, need to declare with pragma:

```c
#pragma omp parallel private (x)
```
OpenMP Programming Model - Review

• Fork - Join Model:

  - OpenMP programs begin as single process (master thread) and executes sequentially until the first parallel region construct is encountered
    - FORK: Master thread then creates a team of parallel threads
    - Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads
    - JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread
#pragma omp parallel num_threads(2)
{
    x=1;
    y=1+x;
}

X=1;
y=1+x;

x=1;
y=1+x;

X and y are shared variables. There is a risk of data race.
#pragma omp parallel
{
    x=1;
    y=1+x;
}

X and y are shared variables. There is a risk of data race.

Assume number of threads=2

Thread 0
X=1;
y=1+x;

Thread 1
x=1;
y=1+x;
X and y are shared variables. There is a risk of data race
Divide for-loop for parallel sections

for (int i=0; i<8; i++) x[i]=0; //run on 4 threads

#pragma omp parallel
{
    int numt=omp_get_num_thread();
    int id = omp_get_thread_num(); //id=0, 1, 2, or 3
    for (int i=id; i<8; i +=numt)
        x[i]=0;
}

// Assume number of threads=4

Thread 0
Id=0;
x[0]=0;
X[4]=0;

Thread 1
Id=1;
x[1]=0;
X[5]=0;

Thread 2
Id=2;
x[2]=0;
X[6]=0;

Thread 3
Id=3;
x[3]=0;
X[7]=0;
Use `pragma parallel for`

```c
for (int i=0; i<8; i++) x[i]=0;
```

```c
#pragma omp parallel for
{
    for (int i=0; i<8; i++)
        x[i]=0;
}
```

*System divides loop iterations to threads*

<table>
<thead>
<tr>
<th>Id=0;</th>
<th>Id=1;</th>
<th>Id=2;</th>
<th>Id=3;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]=0;</td>
<td>x[1]=0;</td>
<td>x[2]=0;</td>
<td>x[3]=0;</td>
</tr>
</tbody>
</table>
OpenMP Data Parallel Construct: Parallel Loop

- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning
- Synchronization also automatic (barrier)

Serial Program:
```c
void main()
{
    double Res[1000];
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Parallel Program:
```c
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
Programming Model – Parallel Loops

• Requirement for parallel loops
  • No data dependencies (reads/write or write/write pairs) between iterations!

• Preprocessor calculates loop bounds and divide iterations among parallel threads

```c
#pragma omp parallel for
for( i=0; i < 25; i++ )
{
    printf(“Foo”);
}
```
```c
for (i=0; i<max; i++) zero[i] = 0;
```

- Breaks *for loop* into chunks, and allocate each to a separate thread
  - *e.g.* if `max = 100` with 2 threads:
    assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - *i.e.* *No* `break, return, exit, goto` statements

In general, don’t jump outside of any pragma block
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<max; i++) { ... }
}
can be shortened to:
#pragma omp parallel for
    for (i=0; i<max; i++) { ... }

• Also works for sections

This is the only directive in the parallel section
Example: Calculating $\pi$

**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 $\Delta x$ and height $F(x_i)$ at the middle of interval $i$. 

**Graphical Illustration:**

- $F(x) = \frac{4.0}{1+x^2}$
- The graph shows a decreasing function from 0 to 1.
- The area under the curve approximates $\pi$. 

**Equation:**

$$F(x) = \frac{4.0}{1+x^2}$$
Sequential Calculation of $\pi$ 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);
}
```
```c
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
    int i; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #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_THREADS)
        {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=1; i<NUM_THREADS; i++)
        sum[0] += sum[i]; pi = sum[0] / num_steps
    printf ("pi = %6.12f\n", pi);
}
```
OpenMP Reduction

double avg, sum=0.0, A[MAX]; int i;
#pragma omp parallel for private ( sum)
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX;  // bug

• Problem is that we really want sum over all threads!
• 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.

double avg, sum=0.0, A[MAX]; int i;
#pragma omp for reduction(+ : sum)
for (i = 0; i <= MAX ; i++)
    sum += A[i];
avg = sum/MAX;
• OpenMP supports reduce operation

```c
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 100; i++) {
    sum += array[i];
}
```

• Reduce ops and init() values (C and C++):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Bitwise</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>0 &amp; ~0</td>
<td>0 &amp; 1</td>
</tr>
<tr>
<td><code>-</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>*</code></td>
<td>1 ^ 0</td>
<td>0</td>
</tr>
</tbody>
</table>
Calculating π Version (1) - review

```c
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
    int i; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #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_THREADS)
        {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=1; i<NUM_THREADS; i++)
        sum[0] += sum[i]; pi = sum[0] / num_steps
    printf("pi = %6.12f\n", pi);
}
```
#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);
}
Loop Scheduling in 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 “barrier” 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?
Impact of Scheduling Decision

• Load balance
  • Same work in each iteration?
  • Processors working at same speed?

• Scheduling overhead
  • Static decisions are cheap because they require no run-time coordination
  • Dynamic decisions have overhead that is impacted by complexity and frequency of decisions

• Data locality
  • Particularly within cache lines for small chunk sizes
  • Also impacts data reuse on same processor
OpenMP environment variables

**OMP_NUM_THREADS**
- sets the number of threads to use during execution
- when dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
- For example,
  ```
  setenv OMP_NUM_THREADS 16 [csh, tcsh]
  export OMP_NUM_THREADS=16 [sh, ksh, bash]
  ```

**OMP_SCHEDULE**
- applies only to `do`/`for` and `parallel do`/`for` directives that have the schedule type `RUNTIME`
- sets schedule type and chunk size for all such loops
- For example,
  ```
  setenv OMP_SCHEDULE GUIDED,4 [csh, tcsh]
  export OMP_SCHEDULE= GUIDED,4 [sh, ksh, bash]
  ```
Programming Model – Loop Scheduling

• schedule clause determines how loop iterations are divided among the thread team
  • static([chunk]) divides iterations statically between threads
    • Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
    • Default [chunk] is $\text{ceil}( \# \text{iterations} / \# \text{threads} )$
  • dynamic([chunk]) allocates [chunk] iterations per thread, allocating an additional [chunk] iterations when a thread finishes
    • Forms a logical work queue, consisting of all loop iterations
    • Default [chunk] is 1
  • guided([chunk]) allocates dynamically, but [chunk] is exponentially reduced with each allocation
## Loop scheduling options

<table>
<thead>
<tr>
<th>static</th>
<th>dynamic (3)</th>
<th>guided (1)</th>
<th>(2)</th>
</tr>
</thead>
</table>
Programming Model – Data Sharing

• Parallel programs often employ two types of data
  • Shared data, visible to all threads, similarly named
  • Private data, visible to a single thread (often stack-allocated)

• PThreads:
  • Global-scoped variables are shared
  • Stack-allocated variables are private

• OpenMP:
  • \texttt{shared} variables are shared
  • \texttt{private} variables are private

// shared, globals
int bigdata[1024];

void* foo(void* bar) {
    int tid;
    #pragma omp parallel
    /\shared \{ bigdata \}
    /\texttt{private} \{ tid \}
    {
        /* Calc. here */
    }
}
Programming Model - Synchronization

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Named or unnamed
    - No *explicit* locks / mutexes
  - Barrier directives
  - Explicit Lock functions
    - When all else fails – may require `flush` directive
  - Single-thread regions *within* parallel regions
    - *master*, *single* directives

```
#pragma omp critical
{
    /* Critical code here */
}

#pragma omp barrier

omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );

#pragma omp single
{
    /* Only executed once */
}
```
int sum=0
#pragma omp parallel for
for(int j=1; j <n; j++){
    int x = j*j;
    #pragma omp critical
    {
        sum=sum+x; // One thread enters the critical section at a time.
    }
}

* May also use

#pragma omp atomic
x += exper

• Faster, but can support only limited arithmetic operation such as
  ++, --, +=, -=, *+, /=, &=, |=
• Elapsed wall clock time:
  
  ```c
  double omp_get_wtime(void);
  ```

  • Returns 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,” so subtract results of two calls to `omp_get_wtime` to get elapsed time
Parallel Matrix Multiply: Run Tasks $T_i$ in parallel on multiple threads

\[
\begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix}
\times
\begin{pmatrix}
5 & 7 \\
6 & 8
\end{pmatrix}
= \begin{pmatrix}
1 \times 5 + 2 \times 6 & 1 \times 7 + 2 \times 8 \\
3 \times 5 + 4 \times 6 & 3 \times 7 + 4 \times 8
\end{pmatrix}
= \begin{pmatrix}
17 & 23 \\
39 & 53
\end{pmatrix}
\]

for $i = 1$ to $n$ do

$T_i$ : for $j = 1$ to $n$ do

\[
\text{sum} = 0;
\]

for $k = 1$ to $n$ do

\[
\text{sum} = \text{sum} + a[i, k] \times b[k, j];
\]

endfor

$c[i, j] = \text{sum}$;

endfor

$T_1$ : Read row $A_i$ and matrix $B$

Write row $C_i$
Parallel Matrix Multiply: Run Tasks $T_i$ in parallel on multiple threads

$$
\begin{pmatrix}
1 & 2 \\
3 & 4 \\
\end{pmatrix} \times 
\begin{pmatrix}
5 & 7 \\
6 & 8 \\
\end{pmatrix} = 
\begin{pmatrix}
1 \times 5 + 2 \times 6 & 1 \times 7 + 2 \times 8 \\
3 \times 5 + 4 \times 6 & 3 \times 7 + 4 \times 8 \\
\end{pmatrix} = 
\begin{pmatrix}
17 & 23 \\
39 & 53 \\
\end{pmatrix}
$$

for $i = 1$ to $n$ do

$T_i$:

for $j = 1$ to $n$ do

$sum = 0$;

for $k = 1$ to $n$ do

$sum = sum + a[i, k] \times b[k, j]$;

endfor

c[i, j] = sum;

endfor

endfor

$T_1$ $T_2$
Matrix Multiply in OpenMP

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

Outer loop spread across N threads; inner loops inside a single thread
OpenMP Summary

• OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code

• OpenMP can enable (easy) parallelization of loop-based code with fork-join parallelism
  
  • `#pragma omp parallel`
  • `#pragma omp parallel for`
  • `#pragma omp parallel private ( i, x )`
  • `#pragma omp atomic`
  • `#pragma omp critical`
  • `#pragma omp for reduction(+ : sum)`

• OpenMP performs comparably to manually-coded threading
  
  • Not a silver bullet for all applications