Agenda

- **Morning**: An Introduction to OpenMP 3.1
- **Afternoon**: Using OpenMP; Hybrid Programming with MPI and OpenMP; OpenMP 4.0
- **Evening**: Practical Programming
Morning Agenda

- What is OpenMP?
- The core elements of OpenMP 3.1
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP usage
  - An example
Massive increase in concurrency within nodes

Node architecture also changing
- Growing core count, heterogeneity, memory size & BW, power attributes, resilience
- Reduced memory per core
- Application codes need to exploit nodes fully
- OpenMP can help
The OpenMP API

- High-level directive-based multithreaded programming
  - User makes strategic decisions; compiler figures out details
  - Shared memory model: Natural fit for shared memory (multicore) platforms, now also heterogeneous systems
  - Can be used with MPI in Fortran, C, C++ programs to reduce memory footprint, communication behavior of MPI code
  - Under active development

```c
#pragma omp parallel
#pragma omp for schedule(dynamic)
for (I=0; I<N; I++){
    NEAT_STUFF(I);
} /* implicit barrier here */
```
OpenMP: Brief History

- Initial version based upon shared memory parallel directive standardization efforts in late 80s
  - PCF and aborted ANSI X3H5
  - Nobody fully implemented either of them
  - Proprietary directives in use for programming early shared memory platforms
- Oriented toward technical computing
  - Fortran, loop parallelism
- Recent work has significantly extended scope of original features
What is OpenMP?

- De-facto standard **API** to write shared memory parallel applications in C, C++, and Fortran
  - Recent features go beyond shared memory
- Initial version released end of 1997
  - For Fortran only
  - Subsequent releases for C, C++
- Version 2.5 merged specs for all three languages
- Version 3.1 released July 2011; 4.0 July 2013
OpenMP 4.0
The OpenMP ARB

- OpenMP is maintained by the OpenMP Architecture Review Board (the ARB), which
  - Interprets OpenMP
  - Writes new specifications - keeps OpenMP relevant
  - Works to increase the impact of OpenMP

- Members are organizations - not individuals
  - Current members
    - Permanent: AMD, Convey Computer, Cray, Fujitsu, HP, IBM, Intel, Microsoft, NEC, Nvidia, Oracle, Red Hat, St Microelectronics, Texas Instruments
    - Auxiliary: ANL, BSC, cOMPunity, EPCC, NASA, LANL, ASC/LLNL, ORNL, RWTH Aachen, SNL, TACC, University of Houston

www.openmp.org
How Does OpenMP Work?

OpenMP provides thread programming model at a “high level”
- Threads collaborate to perform the computation
- They communicate by sharing variables
- They synchronize to order accesses and prevent data conflicts
- Structured programming is encouraged to reduce likelihood of bugs

Alternatives:
- MPI
- POSIX thread library is lower level
- Automatic parallelization is higher level (user does nothing)
  - But usually successful on simple codes only

User makes strategic decisions; Compiler figures out details
# OpenMP 3.1 Components

## Directives
- Parallel region
- Worksharing constructs
- Tasking
- Synchronization
- Data-sharing attributes

## Runtime library
- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Active levels
- Thread limit
- Nesting level
- Ancestor thread
- Team size
- Locking
- Wallclock timer

## Environment variables
- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle threads
- Active levels
- Thread limit

### Pragmas in C / C++

- `#pragma` comments in Fortran
Role of User

- User inserts directives telling compiler how statements are to be executed
  - what parts of the program are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads

- User must remove any dependences in parallel parts
  - Or introduce appropriate synchronization

- OpenMP compiler does not check for them!
  - It is up to programmer to ensure correctness
  - Some tools exist to help check this
How is OpenMP Compiled?

- Most Fortran/C compilers today implement OpenMP
- The user provides the required switch or switches
- Sometimes this also needs a specific optimization level, so manual should be consulted
- May also need to set threads’ stacksize explicitly

Examples
- Commercial: -openmp (Intel, Sun, NEC), -mp (SGI, PathScale, PGI), --openmp (Lahey, Fujitsu), -qsmp=omp (IBM) /openmp flag (Microsoft Visual Studio 2005), etc.
- Freeware: gcc, Omni, OdinMP, OMPi, Open64.UH, (llvm)

Check information at http://www.openmp.org
OpenMP Usage

- If program is compiled sequentially
  - OpenMP comments and pragmas are ignored
- If code is compiled for parallel execution
  - Pragmas drive translation into parallel program
- Ideally, one source for sequential and parallel program (big maintenance plus)
OpenMP Parallel Computing Solution Stack

End User

Application

Prog. Layer (OpenMP API)

Directives, Compiler
OpenMP library
Environment variables

System layer

Compiler’s runtime library

OS/system
Agenda

- What is OpenMP?
- The core elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP usage
  - An example
  - Common programming errors
  - False sharing
OpenMP Fork-Join Execution Model

- Execution starts with a single thread (the initial / master thread)
- Master thread spawns multiple worker threads as needed, together forming a team
- *Parallel region* is a block of code executed by all threads in a team simultaneously

Number of threads in a team may be dynamically adjusted
OpenMP Memory Model

- OpenMP assumes a shared memory model.
- Threads communicate by sharing variables.
- Synchronization protects data conflicts.
- Synchronization is expensive.
- Change how data is accessed to minimize the need for synchronization.

- All threads have access to the same, **globally shared**, memory.
- Data can be shared or private.
- Shared data is accessible by all threads.
- Private data can only be accessed by the thread that owns it.
- Data transfer is transparent to the programmer.
- Synchronization takes place, but it is mostly implicit.
In OpenMP code, data needs to be “labeled”

There are two basic types:

- **Shared** – there is only one instance of the data
  - Threads can read and write the data simultaneously unless protected through a specific construct
  - All changes made are visible to all threads
    - But not necessarily immediately, unless enforced ......

- **Private** - Each thread has a copy of the data
  - No other thread can access this data
  - Changes only visible to the thread owning the data

Data is shared by default
OpenMP Syntax

- Most OpenMP constructs are compiler directives
  - For C and C++, they are pragmas with the form:
    ```c
    #pragma omp construct [clause [clause]...]```
  - For Fortran, the directives may have fixed or free form:
    ```fortran
    !$OMP construct [clause [clause]...]  
    C$OMP construct [clause [clause]...]  
    !$OMP construct [clause [clause]...]```

- Include file and the OpenMP lib module
  ```c
  #include <omp.h>  
  use omp_lib```

- Most OpenMP constructs apply to a "structured block".
  - A block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It’s OK to have an exit() within the structured block.

OpenMP sentinel forms: `#pragma omp  !$OMP`
Example - The Reduction Clause

```
sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
  do i = 1, n
    sum = sum + x(i)
  end do
!$omp end do
!$omp end parallel
print *,sum
```

- The result is available after the parallel region
- The compiler generates optimized code that enables threads to collaborate to perform the reduction
- The reduction can be hidden in a function call

`reduction ( operator: list )`
Agenda

- What is OpenMP?
- The core elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP usage
  - An example
  - Common programming errors
  - False sharing
Defining Parallelism In OpenMP

- A parallel region is a block of code executed by all threads in a team simultaneously
  - Threads in team are numbered consecutively, starting from 0; the master thread has thread ID 0
  - Thread adjustment (if enabled) is only done before entering a parallel region
  - Parallel regions can be nested, but support for this is implementation dependent
  - An "if" clause can be used to guard the parallel region; if the condition evaluates to "false", the code is executed serially

OpenMP Team := Master + Workers
Parallel Regions

- You create a team of threads in OpenMP with the "omp parallel" pragma.
- For example, to create a 4 thread parallel region:

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

- **Each thread executes a copy of the code within the structured block**
- **Each thread calls** pooh(ID,A) **for ID = 0 to 3**
Parallel Regions

- Each thread executes the same code redundantly.

```c
#include <omp.h>

int main() {
    double A[1000];
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        pooh(ID, A);
    }
    printf("all done\n");
    return 0;
}
```

A single copy of A is shared between all threads. Threads wait here for all threads to finish before proceeding (i.e., a barrier).
Parallel Regions and The “if” Clause

Active vs. inactive parallel regions.

- An optional \texttt{if} clause causes the parallel region to be active only if the logical expression within the clause evaluates to true.
- An if clause that evaluates to false causes the parallel region to be inactive (i.e. executed by a team of size one).

```c
double A[N];

#pragma omp parallel if(N>1000)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```
Scope of OpenMP Region

- A parallel region can span multiple source files.

```c
foo.f
C$OMP PARALLEL
  call whoami
C$OMP END PARALLEL

bar.f
subroutine whoami
  external omp_get_thread_num
  integer iam, omp_get_thread_num
  iam = omp_get_thread_num()
C$OMP CRITICAL
  print*, ’Hello from ’, iam
C$OMP END CRITICAL
return
end
```

- **Lexical extent** of parallel region includes lexical extent.
- **Dynamic extent** of parallel region includes lexical extent.

Orphaned directives can appear outside a parallel construct.
A Multi-threaded “Hello world” Program

- Each thread prints “hello world” in no specific order

```c
#include "omp.h"
void main()
{
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) 
", ID);
}
}
```

Sample Output:

- hello(1) hello(0) world(1)
- world(0)
- hello (3) hello(2) world(2)
- world(3)
**Example: The PI Program**

**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 \( i \).
Pi Program: Sequential Version

```c
#define NUMSTEPS 100000000
double step;
void main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) NUMSTEPS;

    for (i=1;i<= NUMSTEPS; i++) {
        x = (i-0.5)*step;
        sum += 4.0/(1.0+x*x);
    }

    pi = step * sum;
}
```
Get the exercise codes

Download the exercises with:

```
$ wget http://www.cs.uh.edu/~dreachem/ATPESC14-omp-exercises.tar.gz
```

To run an OpenMP program on 1 node with, e.g., 8 threads:

```
$ runjob --block $COBALT_PARTNAME -- p 1 --np 1 \ 
--envs OMP_NUM_THREADS=8 : ./omp-program
```
Exercise: Parallel Pi

Create a parallel version of the Pi program. Output time and number of threads used, for small numbers of threads.

- Use the parallel construct. Pay close attention to shared versus private variables.

- In addition to a parallel construct, you should use these runtime library routines:
  - int omp_get_num_threads(); Get / set number of threads in team
  - void omp_set_num_threads();
  - int omp_get_thread_num(); Get thread ID (rank)
  - double omp_get_wtime(); Time in sec since fixed point in past
Exercise: OpenMP Pi Program

#include <omp.h>
static long num_steps = 100000000;
double step;
#define NUM_THREADS 8
void main ()
{
    int i, nthreads; double x, pi, sum[NUM_THREADS] ={0};
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        double x; int id, i, nthrds;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id;i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads; i++)pi += sum[i] * step;
}
Comparison with MPI: Pi program

```c
#include <mpi.h>
void main (int argc, char *argv[])
{
    int i, my_id, numprocs; double x, pi, step, sum = 0.0;
    step = 1.0/(double) num_steps;
    MPI_Init(&argc, &argv);
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);
    my_steps = num_steps/numprocs;
    for (i=my_id*my_steps; i<(my_id+1)*my_steps; i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
}
```
OpenMP and MPI

Next Improvements:
- more flexible worksharing construct?
- Optimize use of cache
Agenda

- What is OpenMP?
- The core elements of OpenMP
  - Parallel regions
  - Work-sharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP usage
  - An example
## Worksharing Constructs

<table>
<thead>
<tr>
<th>Sequential code</th>
<th>OpenMP parallel region</th>
<th>OpenMP parallel region and a worksharing for construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i=0; i&lt;N; i++) { a[i] = a[i] + b[i]; }</td>
<td>#pragma omp parallel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
<td>#pragma omp parallel</td>
</tr>
<tr>
<td></td>
<td>int id, i, Nthrds, istart, iend;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>id = omp_get_thread_num();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Nthrds = omp_get_num_threads();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>istart = id * N / Nthrds;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>iend = (id+1) * N / Nthrds;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for(i=istart; i&lt;iend; i++) { a[i] = a[i] + b[i]; }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>#pragma omp parallel</td>
</tr>
<tr>
<td></td>
<td>#pragma omp for schedule(static)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for(i=0; i&lt;N; i++) { a[i] = a[i] + b[i]; }</td>
<td></td>
</tr>
</tbody>
</table>
OpenMP Worksharing Constructs

- Divides the execution of the enclosed code region among the members of the team
- The “for” worksharing construct splits up loop iterations among threads in a team
- Each thread gets one or more “chunks”

```c
#pragma omp parallel
#pragma omp for
for (i = 0; i < N; i++) {
    work(i);
}
```

By default, all threads wait at a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.

```c
#pragma omp for nowait
```

“nowait” is useful between two consecutive, independent omp for loops.

`omp do` in Fortran
Example: OMP For

```c
#pragma omp parallel default(none)\ 
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

} /*-- End of parallel region --*/

(implied barrier)
```
Example: A Linked List

while(my_pointer) {
    (void) do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop

Loops must be countable. To parallelize this loop, it is necessary to first count the number of iterations and then rewrite it as a \texttt{for} loop. More on this later…
Loop Collapse

- Allows parallelization of perfectly nested loops without using nested parallelism
- The collapse clause on for/do loop indicates how many loops should be collapsed
- The compiler forms a single loop and parallelizes it

```
!$omp parallel do collapse(2) ...
do i = il, iu, is
   do j = jl, ju, js
      do k = kl, ku, ks
          ....
          end do
      end do
end do
end do
!$omp end parallel do
```
## OpenMP Schedule Clause

The schedule clause affects how loop iterations are mapped onto threads.

```
schedule ( static | dynamic | guided [, chunk] )
schedule ( auto | runtime )
```

| **static** | Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion |
| **dynamic** | Fixed portions of work; size is controlled by the value of chunk. When a thread finishes, it starts on the next portion of work |
| **guided** | Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially |
| **auto** | The compiler (or runtime system) decides what is best to use; choice could be implementation dependent |
| **runtime** | Iteration scheduling scheme is set at runtime via environment variable `OMP_SCHEDULE` or runtime library call |
Example Of a Static Schedule

A loop of length 16 using 4 threads

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>no chunk *</td>
<td>1-4</td>
<td>5-8</td>
<td>9-12</td>
<td>13-16</td>
</tr>
<tr>
<td>chunk = 2</td>
<td>1-2</td>
<td>3-4</td>
<td>5-6</td>
<td>7-8</td>
</tr>
<tr>
<td></td>
<td>9-10</td>
<td>11-12</td>
<td>13-14</td>
<td>15-16</td>
</tr>
</tbody>
</table>

*) The precise distribution is implementation defined
500 Iterations, 4 Threads

Thread ID

Iteration Number

- guided, 5
- dynamic, 5
- static
# The Schedule Clause

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
</tbody>
</table>

- **STATIC**: scheduling done at compile-time
- **DYNAMIC**: complex scheduling logic used at run-time
- **GUIDED**: least work at runtime; scheduling done at compile-time

Least work at runtime: scheduling done at compile-time
Most work at runtime: complex scheduling logic used at run-time
OpenMP Sections

- Work-sharing construct
- Gives a different structured block to each thread

```c
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
  x_calculation();
#pragma omp section
  y_calculation();
#pragma omp section
  z_calculation();
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.
Example: Overlap I/O, Processing

```c
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) read_input(i);
            (void) signal_read(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_read(i);
            (void) process_data(i);
            (void) signal_processed(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_processed(i);
            (void) write_output(i);
        }
    }
} /*-- End of parallel sections --*/
```
# Overlap I/O And Processing

<table>
<thead>
<tr>
<th>Time</th>
<th>Input Thread</th>
<th>Processing Thread(s)</th>
<th>Output Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
OpenMP Master

- Denotes a structured block executed by the master thread
- The other threads just skip it
  - no synchronization is implied

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
#pragma omp master
    {   exchange_boundaries();   }
#pragma barrier
    do_many_other_things();
}
```
OpenMP Single

- Denotes a block of code that is executed by only one thread.
- A barrier is implied at the end of the single block.

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```
Combined Parallel/Work-share

- OpenMP shortcut: Put the “parallel” and the work-share on the same line

```c
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

These are equivalent

```c
double res[MAX]; int i;
#pragma omp parallel for
for (i=0;i< MAX; i++) {
    res[i] = huge();
}
```

- There’s also a “parallel sections” construct.
Recall: The OpenMP specification does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be orphaned.

They can appear outside the lexical extent of a parallel region.
More On Orphaning

When an orphaned worksharing or synchronization directive is encountered in the sequential part of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored.

```
(void) dowork();  // Sequential FOR
#pragma omp parallel
{
    (void) dowork();  // Parallel FOR
}
```
Exercise 2:

- Modify your program that uses numerical integration to compute an estimate of PI.
- This time, use a work-sharing construct
- Remember, you’ll need to make sure multiple threads don’t overwrite each other’s variables.
OpenMP “SPMD” PI Program

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int I, nthreads; double x, pi, sum[NUM_THREADS] = 0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        double x; int id, i, nthrds;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id;i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

SPMD: Each thread runs the same code. The thread ID enables thread-specific behavior.

- Promote scalar so each thread computes own portion of result
- To avoid data race, one thread copies value to global variable
- Creates cyclic distribution of iterations to threads
#include <omp.h>
static long num_steps = 100000;   double step;
#define NUM_THREADS 2
void main ()
{
    int i;   double x, pi, sum[NUM_THREADS] ={0.0};
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        double x;   int i, id;
        id = omp_get_thread_num();
        #pragma omp for
        for (i=0;i< num_steps; i++)
        {  
            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] * step;
OpenMP PI Program with Reduction

#include <omp.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 reduction(+:sum) private(x)
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}

OpenMP adds 1-2 lines of code
#include <stdlib.h>
#include <sys/time.h>

void * compute_pi(void *dat)
{
    int threadid = ((thr_data_t*)dat)->threadid;
    int num_threads = ((thr_data_t*)dat)->num_threads;
    int num_steps = ((thr_data_t*)dat)->num_steps;
    pthread_mutex_t *mtx = ((thr_data_t*)dat)->mtx;
    double *sump = ((thr_data_t*)dat)->sump;

    double step = 1.0 / num_steps;
    local_sum = 0.0;
    /* round robin distribution of iterations */
    for (i = threadid; i < num_steps; i += num_threads) {
        x = (i - 0.5)*step;
        local_sum += 4.0 / (1.0 + x*x);
    }

    pthread_mutex_lock(mtx);
    *sump = *sump + local_sum;
    pthread_mutex_unlock(mtx);
    return NULL;
}

int main(int argc, char **argv)
{
    /* start pi calculation */
    threads = malloc(num_threads * sizeof *threads);
    step = 1.0 / num_steps;
    pthread_mutex_init(&mtx, NULL);

    /* spawn threads to work on computing pi */
    for (i = 0; i < num_threads; i++) {
        dat[i].threadid = i;
        dat[i].num_threads = num_threads;
        dat[i].num_steps = num_steps;
        dat[i].mtx = &mtx;
        dat[i].sump = ∑
        pthread_create(&threads[i], NULL, compute_pi,
                       (void *)&dat[i]);
    }

    /* join threads */
    for (i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    pi = step * sum;
    free(dat);
    pthread_mutex_destroy(&mtx);
    free(threads);
    ...
}

Requires explicit thread/data management
OpenMP and MPI

Calculating Pi: Comparing Pthreads, OpenMP, and MPI on dual-socket Intel Xeon E5-2665

- **pthreads**
- **omp v1**
- **omp v2 (omp for)**
- **omp v3 (omp for reduction)**
- **mpi**

The graph shows the time (in seconds) required to calculate Pi as a function of the number of processors (# procs) for different parallelization methods.
Agenda

- What is OpenMP?
- The core elements of OpenMP
  - Parallel regions
  - Work-sharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP usage
  - An example
OpenMP Synchronization

- Synchronization enables the user to
  - Control the \textit{ordering of executions} in different threads
  - Ensure that at most one thread executes operation or region of code at any given time (\textit{mutual exclusion})

- High level synchronization:
  - barrier
  - critical section
  - Atomic
  - ordered

- Low level synchronization:
  - flush
  - locks (both simple and nested)
Barrier

When these loops are parallelized, we need to be sure to **update all of a[] before using a[]** *

```c
for (i=0; i < N; i++)
    a[i] = b[i] + c[i];
```

**wait !**

```c
for (i=0; i < N; i++)
    d[i] = a[i] + b[i];
```

All threads wait at the barrier point and only continue when all threads have reached the barrier point

*) If the mapping of iterations onto threads is guaranteed to be identical for both loops, we do not need to wait. This is the case with the static schedule under certain conditions*
Barrier syntax in OpenMP:

```plaintext
#pragma omp barrier
#$omp barrier
```
Barrier: Explicit and Implicit

- Each thread waits until all threads arrive.

```c
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
    for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
    #pragma omp for nowait
    for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
    A[id] = big_calc3(id);
}
```

- Implicit barrier at the end of a parallel region.
- Implicit barrier at the end of a `for` work-sharing construct.
- No implicit barrier due to `nowait`.
The Nowait Clause

- Barriers are implied at end of parallel region, for/do, sections and single constructs
- Barrier can be suppressed by using the optional nowait clause
  - If present, threads do not synchronize/wait at the end of that particular construct

```
#pragma omp for nowait
{
    :
}
```
```
!$omp do :
    :
!$omp end do nowait
```
Mutual Exclusion

- Code may only be executed by at most one thread at any given time
- Could lead to long wait times for other threads
  - Atomic updates for individual operations
  - Critical regions and locks for structured regions of code

![Critical Region Diagram]
Critical Region (Section)

- Only one thread at a time can enter a critical region

```c
float res;
#pragma omp parallel
{
    float B;  int i;
    #pragma omp for
    for(i=0;i<niters;i++){
        B = big_job(i);
    }
    #pragma omp critical
    consume (B, RES);
}
```

Threads wait their turn – only one at a time calls consume()

Use e.g. when all threads update a variable and the order in which they do so is unimportant. Preserves data integrity.
Atomic

- **Atomic** is a special case of mutual exclusion
- It applies only to the update of a memory location

```c
C$OMP PARALLEL PRIVATE(B)
    B = DOIT(I)
    tmp = big_ugly();
C$OMP ATOMIC
    X = X + temp
C$OMP END PARALLEL
```

OpenMP 3.1 describes the behavior in more detail via these clauses:
- read, write, update, capture

The pre-3.1 atomic construct is equivalent to

```
#pragma omp atomic update
```
Ordered

- The **ordered** construct enforces the sequential order for a block.
- Code is executed in order in which iterations would be performed sequentially.

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered
for (i=0;i<N;i++){
    tmp = NEAT_STUFF(i);
    #pragma ordered
    res += consum(tmp);
}
```
Updates to Shared Data

- Blocks of data are fetched into cache lines
- Values may temporarily differ from other copies of data within a parallel region

Diagram:

- Shared memory
- Cache lines (cache1, cache2, ..., cacheN)
- Processes (proc1, proc2, ..., procN)
If shared variable X is kept within a register, the modification may not be immediately visible to the other thread(s).
The Flush Directive

- Flushing is what creates a consistent view of shared data: it causes a thread to write data back to main memory and retrieve new values of updated variables.
- It is automatically performed on a number of constructs.
- The `flush` construct allows the programmer to define a point where a thread makes its variable values consistent with main memory.
  - Caution: it does not enable a thread to retrieve values updated by another thread unless that thread also performs a flush.
  - It also does not synchronize threads.
  - Its use is tricky: be sure you understand it.
The Flush Directive

- Flush also enforces an ordering of memory operations
- When the flush construct is encountered by a thread
  - All memory operations (both reads and writes) defined prior to the sequence point must complete.
  - All memory operations (both reads and writes) defined after the sequence point must follow the flush.
  - Variables in registers or write buffers must be updated in memory.
- Arguments to flush specify which variables are flushed.
- If no arguments are specified, all thread visible variables are flushed.

Use of flush with explicit variables strongly discouraged
What Else Does Flush Influence?

The flush operation does not actually synchronize different threads. It just ensures that a thread’s values are made consistent with main memory.

Compilers reorder instructions to better exploit the functional units and keep the machine busy

- Flush prevents the compiler from doing the following:
  - Reorder read/writes of variables in a flush set relative to a flush.
  - Reorder flush constructs when flush sets overlap.

- A compiler CAN do the following:
  - Reorder instructions NOT involving variables in the flush set relative to the flush.
  - Reorder flush constructs that don’t have overlapping flush sets.
A Flush Example

Pair-wise synchronization.

```plaintext
integer ISYNC(NUM_THREADS)
C$OMP PARALLEL DEFAULT (PRIVATE) SHARED (ISYNC)
   IAM = OMP_GET_THREAD_NUM()
   ISYNC(IAM) = 0
C$OMP BARRIER
   CALL WORK()
   ISYNC(IAM) = 1
C$OMP FLUSH(ISYNC)
   DO WHILE (ISYNC(NEIGHBOR) .EQ. 0)
C$OMP FLUSH(ISYNC)
   END DO
C$OMP END PARALLEL
```

Make sure other threads can see my write.

! I’m done; signal this to other threads

Make sure the read picks up a good copy from memory.
Implied Flush

Flushes are implicitly performed during execution:

- In a *barrier* region
- At *exit from* worksharing regions, unless a nowait is present
- At *entry to and exit from* parallel, critical, ordered and parallel worksharing regions
- During `omp_set_lock` and `omp_unset_lock` regions
  - During `omp_test_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock` and `omp_test_nest_lock` regions, if the region causes the lock to be set or unset
- Immediately *before and after* every task scheduling point
- At *entry to and exit from* atomic regions, where the list contains only the variable updated in the atomic construct
- But *not* on entry to a worksharing region, or entry to/exit from a master region,