COSC 6374
Parallel Computation

Introduction to OpenMP (II)

Material is based on slides
by Barbara Chapman (UH) and Tim Mattson (Intel)

Edgar Gabriel
Fall 2011

Agenda

• Parallel computing, threads, and OpenMP
• The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
What are Threads?

- Thread: an independent flow of control
  - Runtime entity created to execute sequence of instructions
- Threads require:
  - A program counter
  - A register state
  - An area in memory, including a call stack
  - A thread id
- A process is executed by one or more threads that share:
  - Address space
  - Attributes such as UserID, open files, working directory, etc.

OpenMP

- Provides thread programming model at a “high level”.
  - The user does not need to specify all the details
  - Especially with respect to the assignment of work to threads
  - Creation of threads
- User makes strategic decisions
  - Compiler figures out details
OpenMP Overview: How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program’s outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

Syntax details

- Most of the constructs in OpenMP are compiler directives.
- For C and C++:
  - Directives are pragmas with the form:
    
    ```
    #pragma omp construct [clause [clause]...]
    
    Include file: #include <omp.h>
    ```

- For Fortran:
  - the directives are comments and take one of the forms:
    - Fixed form
      
      ```
      C$OMP construct [clause [clause]...]
      ```
    - Free form (but works for fixed form too)
      
      ```
      !$OMP construct [clause [clause]...]
      ```
    - The OpenMP lib module: use omp_lib
Structured blocks (C/C++)

- Most OpenMP constructs apply to structured blocks.
  - Structured block: a block with one point of entry at the top and one point of exit at the bottom.
  - The only “branches” allowed are `STOP` statements in Fortran and `exit()` in C/C++.
  - In C/C++: a block is a single statement or a group of statements between brackets

```
#pragma omp parallel
{
    id = omp_get_thread_num();
    res[id] = do_work(id);
}
```

```
#pragma omp for
for(i=0; i<N; i++){
    res[i] = big_calc(i);
    A[i] = B[i] + res[i];
}
```

- **Structured Block Boundaries**

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job(id);
    if(!conv(res[id]) goto more;
}
printf(" All done \n");
```

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job(id);
    if(conv(res[id]) goto done;
        goto more;
}
done: if(!really_done())
    goto more;
```

- **A structured block**
- **Not a structured block**
Parallel Computation

Edgar Gabriel

Parallel Regions

• Threads are created using `omp parallel` pragma
• Each thread executes a copy of the code within the structured block
• How many threads are created?
  - Environment variable to set no. of threads
  - Runtime function
  - System ‘default’ used if no additional information given

```c
double A[1000];
#pragma omp parallel
{
    // do some work
}
printf("all done\n");
```

Parallel Computation

Edgar Gabriel

Parallel Regions

• Fork-join model of OpenMP
  - Threads are created at the beginning of a parallel region and destroyed at the end of the parallel region (conceptually)

```
double A[1000];
pooh(0,A) pooh(1,A) pooh(2,A) pooh(3,A)
printf("all done\n"); Threads wait here for all threads to finish before proceeding (i.e. a barrier)
```
A multi-threaded “Hello world” program

- Starting point: sequential “hello world”

```c
int main(int argc, char **argv)
{
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
```

- Sample output

```c
#include "omp.h"
int main( int argc, char **argv) {
    #pragma omp parallel
    {
        int ID= omp_get_thread_num();
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    }
    return 0;
}
```

```plaintext
hello(1) hello(0) world(1)
world(0)
hello(3) hello(2) world(3)
world(2)
```
OpenMP Library routines

- **Modify/Check the number of threads**
  
  - `omp_set_num_threads()`
  - `omp_get_num_threads()`
  - `omp_get_thread_num()`
  - `omp_get_max_threads()`

- **Are we in a parallel region?**
  
  - `omp_in_parallel()`

- **How many processors in the system?**
  
  - `omp_num_procs()`

---

Example: vector add operation

**Sequential code**

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

**OpenMP parallel version**

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int Nthrds = omp_get_num_threads();
    int istart = id * N / Nthrds;
    int iend = (id+1) * N / Nthrds;

    for (i=istart; i<iend; i++) {
        a[i] = a[i] + b[i];
    }
}
```
Example: vector add operation

- All variables declared inside of the parallel region are considered to be private to each thread
  - Each thread has its own ‘copy’ of the variable
- Variables declared outside of a parallel region are shared amongst threads
  - Unless explicitly changed by the user

- If \texttt{istart} and \texttt{iend} are not coordinated ‘cautiously’, it can lead to cache coherence problems
  - \texttt{e.g. a[iend]} on thread with id=x and \texttt{a[istart]} on the thread with id=x+1 are in the same cache line

OpenMP work-sharing constructs

- The \texttt{for} work-sharing construct splits up loop iterations among the threads

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

- By default, there is a barrier at the end of the \texttt{omp for}. Use the \texttt{nowait} clause to turn off the barrier, \texttt{e.g.}

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

- \texttt{nowait} is useful between two consecutive, independent \texttt{omp for} loops.
Example: vector add operation

```c
#pragma omp parallel
#pragma omp for
for ( i=0; i<N; i++ ) {
    a[i] = a[i] + b[i];
}
```

- Much simpler code than the previous OpenMP version
- Loop variable `i` is automatically declared to be private on each thread
- Does not define how the loop iterations are distributed among the threads
- Can use the schedule clause to influence the work distribution, e.g.
  ```c
  #pragma omp for schedule(static)
  ```

OpenMP for construct

- The schedule clause affects how loop iterations are mapped onto threads
  - `schedule(static [,chunk])`
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - `schedule(dynamic[,chunk])`
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - `schedule(guided[,chunk])`
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - `schedule(runtime)`
    - Schedule and chunk size taken from the `MP_SCHEDULE` environment variable.
OpenMP 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>

Least work at runtime: scheduling done at compile-time

Most work at runtime: complex scheduling logic used at run-time

OpenMP work-sharing constructs

- The sections work-sharing construct gives a different structured block to each thread.
- By default, there is a barrier at the end of the omp sections. Use the nowait clause to turn off the barrier.

```c
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
    X_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
OpenMP work-sharing constructs

- The *master* construct denotes a structured block that is only executed by the master thread. The other threads just skip it.
- No synchronization is implied at the end of the master block.

```c
#pragma omp parallel
{
    do_many_things();
    #pragma omp master
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```

OpenMP work-sharing constructs

- The *single* construct 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
{
    do_many_things();
    #pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
Combined parallel/work-share constructs

- 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();
}
```

Data Environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are `SHARED` among threads
  - C: File scope variables, static
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are `PRIVATE`
  - Automatic variables within a statement block are `PRIVATE`. 
**Data Sharing Examples**

```c
double A[10];
int index[10];

#pragma omp parallel
{
    work(index);
}
printf("%d\n",index[0]);
```

void work (int * index)
{
    float temp[10];
    int count;
    ...
    return;
}

**Data Environment:**

**Changing storage attributes**

- One can selectively change storage attributes constructs using the following clauses*
  - `SHARED`
  - `PRIVATE`
  - `FIRSTPRIVATE`
  - `THREADPRIVATE`
  - The value of a private inside a parallel loop can be transmitted to a global value outside the loop with:
    - `LASTPRIVATE`
  - The default status can be modified with:
    - `DEFAULT (PRIVATE | SHARED | NONE)`

* Third party trademarks and names are the property of their respective owner.
Private Clause

- `private(var)` creates a local copy of `var` for each thread
  - The value is uninitialized
  - Private copy is **not** storage-associated with the original
  - The original is undefined at the end

```c
int var = 13;
#pragma omp parallel for private (var)
for ( j=0; j<1000; j++ ) {
    var = var + j;
}
printf ("%d
", var);
```

- Each thread gets its own `var` which are however not initialized
- Regardless of initialization, `var` is undefined at the end of the parallel region

Firstprivate Clause

- `firstprivate` is a special case of `private`.
  - Initializes each private copy with the corresponding value from the master thread.

```c
int var = 13;
#pragma omp parallel for firstprivate (var)
for ( j=0; j<1000; j++ ) {
    var = var + j;
}
printf ("%d
", var);
```

- Each thread gets its own `var` with an initial value of 13
- Regardless of initialization, `var` is undefined at the end of the parallel region
Lastprivate Clause

- Lastprivate passes the value of a private from the last iteration to a global variable.

```c
int var = 13;
#pragma omp parallel for firstprivate (var) lastprivate(var)
for ( j=0; j<1000; j++ ) {
    var = var + j;
}
printf ("%d\n", var);
```

- Each thread gets its own var with an initial value of 13
- var is defined as its value at the “last sequential” iteration (i.e. for j=999)

OpenMP: A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```c
int A=1, B=1, C=1;
#pragma omp parallel private(B) firstprivate (C)
Inside this parallel region ...
    - A is shared by all threads; equals 1
    - B and C are local to each thread.
        - B’s initial value is undefined
        - C’s initial value equals 1
Outside this parallel region ...
    - The values of B and C are undefined.
```