Accelerate Your Science: An Introduction to High Performance Computing
Lecture 3: Introduction to OpenMP

Guido Juckeland
Visiting Scholar
Technische Universität Dresden, Germany

July 8, 2014
Overview

• OpenMP is an API for writing multithreaded applications
  – A set of compiler directives and library routines for parallel application programmers
  – Greatly simplifies writing multithreaded programs in Fortran, C and C++
  – De-facto standard for shared memory programming

• All slides taken from Tim Mattson: “A ‘Hands-on’ Introduction to OpenMP” (http://openmp.org/wp/resources/ #Tutorials)
Core Syntax

• Most of the constructs in OpenMP are compiler directives.
  
  \#pragma omp construct [clause [clause]…]

  • Example
    
    \#pragma omp parallel num_threads(4)

• Function prototypes and types in the file:
  
  \#include <omp.h>

• Most OpenMP constructs apply to a “structured block”.
  – 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.
How do threads interact?

• OpenMP is a multi-threading, shared address 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.
Programming model

- Fork-Join Parallelism:
  - Master thread spawns a team of threads as needed.
  - Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.
Thread creation: Parallel regions

- You create threads in OpenMP* with the `parallel` construct.
- 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 calls `pooh(ID,A)` for `ID = 0` to `3`
You create threads in OpenMP* with the `parallel` construct.

For example, to create a 4 thread parallel region:

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

Each thread executes a copy of the code within the structured block clause to request a certain number of threads.

Runtime function returning a thread ID.

Each thread calls `pooh(ID, A)` for `ID = 0` to `3`
Thread Creation: Parallel Regions

- Each thread executes the same code redundantly.

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

A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a barrier)
OpenMP: what the compiler does

- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left.
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for each parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.
Synchronization

- Synchronization: bringing one or more threads to a well defined and known point in their execution.
- The two most common forms of synchronization are:

**Barrier:** each thread wait at the barrier until all threads arrive.

**Mutual exclusion:** Define a block of code that only one thread at a time can execute.
Types of synchronization

• High level synchronization:
  – critical
  – atomic
  – barrier
  – ordered

• Low level synchronization
  – flush
  – locks (both simple and nested)

Synchronization is used to impose order constraints and to protect access to shared data.
Barrier

- Each thread waits until all threads arrive (Caution!)

```c
#pragma omp parallel
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    B[id] = big_calc2(id, A);
}
```
Critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

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

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

- **Atomic** provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example).

```c
#pragma omp parallel
{
    double tmp, B;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}
```

The statement inside the atomic must be one of the following forms:
- `x binop= expr`
- `x++`
- `++x`
- `x--`
- `--x`

X is of scalar type and `binop` is a non-overloaded built-in operator.
SPMD vs. worksharing

- A parallel construct by itself creates an SPMD or “Single Program Multiple Data” program (i.e., each thread redundantly executes the same code).
- How do you split up pathways through the code between threads within a team?
  - This is called worksharing
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct
The loop worksharing constructs

- The loop worksharing construct splits up loop iterations among the threads in a team.

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

**Loop construct name:**
- C/C++: `for`
- Fortran: `do`

The variable `I` is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause.
Loop constructs vs. manual parallel regions

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

- **Loop construct:**
  
  ```c
  #pragma omp parallel for
  for(i=0;i< N;i++)
    a[i] = a[i] + b[i];
  ```

- **Manual parallel region**
  
  ```c
  #pragma omp parallel
  {
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds=omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1)iend = N;
    for(i=istart;i<iend;i++)
      a[i] = a[i] + b[i];
  }
  ```
Scheduling work sharing

• 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 OMP_SCHEDULE environment variable (or the runtime library).
  – schedule(auto)
    • Schedule is left up to the runtime to choose (does not have to be any of the above).
Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```c
int i, j, A[MAX];
j = 5;
for (i=0; i< MAX; i++) {
    j += 2;
    A[i] = big(j);
}
```

```c
#pragma omp parallel for
for (i=0; i< MAX; i++) {
    int j = 5 + 2*(i+1);
    A[i] = big(j);
}
```
Nested loops

• For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```c
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
    for (int j=0; j<M; j++) {
        ......
    }
}
```

• Will form a single loop of length N x M and then parallelize that.
• Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.
Reduction

• How do we handle this case?

```c
double ave=0.0, A[MAX]; int i;
for (i=0; i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```

• We are combining values into a single accumulation variable (ave) ... there is a true dependence between loop iterations that can’t be trivially removed

• This is a very common situation ... it is called a “reduction”.

• Support for reduction operations is included in most parallel programming environments.
OpenMP reductions

- **OpenMP reduction clause:** `reduction (op : list)`
- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```c
double ave=0.0, A[MAX]; int i;
#pragma omp parallel for reduction(+:ave)
for (i=0;i< MAX; i++) ave += A[i];
ave = ave/MAX;
```
Implicit barriers and nowait

```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_calc4(id);
}
```

- Implicit barrier at end of workshare construct
- No barrier due to nowait
- Implicit barrier at end of parallel region
Master Construct

- 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).

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

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a **nowait** clause).

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

    do_many_other_things();
}
```
Section construct

- The sections worksharing 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();
}
}
```
Data environment: Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are **SHARED** among threads
  - Fortran: **COMMON** blocks, **SAVE** variables, **MODULE** variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (**ALLOCATE**, **malloc**, **new**)
- But not everything is shared...
  - Stack variables in **subprograms** (Fortran) or **functions** (C) called from parallel regions are **PRIVATE**
  - Automatic variables within a statement block are **PRIVATE**.
double A[10];
int main() {
    int index[10];
#pragma omp parallel
    work(index);
    printf("%d\n", index[0]);
}

extern double A[10];
void work(int *index) {
    double temp[10];
    static int count;
    ...
}

A, index, count

temp
temp
temp

A, index, count

A, index and count are shared by all threads.
temp is local to each thread
Changing storage attributes

• One can selectively change storage attributes for constructs using the following clauses (All data clauses apply to parallel constructs and worksharing constructs except “shared” which only applies to parallel constructs.)
  – SHARED
  – PRIVATE
  – FIRSTPRIVATE

• The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  – LASTPRIVATE
Private clause

- `private(var)` creates a new local copy of `var` for each thread.
  - The value of the private copies is uninitialized
  - The value of the original variable is unchanged after the region

```c
void wrong() {
    int tmp = 0;
    #pragma omp parallel for private(tmp)
    for (int j = 0; j < 1000; ++j)
        tmp += j;
    printf("%d\n", tmp);
}
```

`tmp` was not initialized!!

`tmp` is 0 here!!
Firstprivate clause

- Variables initialized from shared variable
- C++ objects are copy-constructed

```cpp
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
    if ((i%2)==0) incr++;
    A[i] = incr;
}
```

Each thread gets its own copy of `incr` with an initial value of 0
Lastprivate clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```c
void sq2(int n, double *lastterm)
{
    double x; int i;
    #pragma omp parallel for lastprivate(x)
    for (i = 0; i < n; i++){
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
    }
    *lastterm = x;
}
```

“x” has the value it held for the “last sequential” iteration (i.e., for i=(n-1))
Wrap up

- OpenMP used to describe parallelism and cooperation in a program
- Compiler does the work
- Exploits all cores on a node