Tutorial 03 – OpenMP
ECE 459: Programming for Performance

Husam Suleiman
hsuleima@uwaterloo.ca

University of Waterloo

February 10, 2015

Basic Functions and Data Environment

- Some basic functions:
  - omp_get_thread_num(): returns the thread ID
  - omp_get_num_threads(): returns the number of threads in parallel region currently in the team executing the parallel region
  - omp_get_max_threads(): returns the maximum number of threads in a parallel region
  - omp_set_thread_num(int num_threads): sets the number of threads used in the next parallel region

- Data environment
  - Shared

    Shared variable exists in a memory location that is accessible by all threads to read from or write to it. A programmer must ensure a proper access to the shared variable to avoid race condition because all threads access the same address space

  - Private

    Each thread has a local copy and uses it as a temporary variable. The original object is not associated with any others. Changes are only visible to thread owning the data – others can't access it.

  - Default (none|private|shared)

    When using *none* as a default, the programmer needs to explicitly scope all variables. Only one default clause can be identified on a parallel region.

Synchronization Directives

- Master: a region of code that is to be executed by ONLY the master thread of the team. Other threads on the team skip this region of code
  - #pragma omp master
    structured_block
- Critical: only one thread at a time can enter the critical region of code
  - #pragma omp critical
    structured_block
- Atomic: it's a special case of critical. It *only* implies to the update of a memory location atomically by one thread
  - #pragma omp atomic
    statement_expression
- Ordered: it enforces the sequential order for a block
  - #pragma omp ordered
    structured_block
- Barrier: each thread waits until all threads arrive
  - #pragma omp barrier

Synchronization Directives

- Both are equivalent

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

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

```
const int m = 100;
float x[m], y[m], a=0.5;
int i;
#pragma omp parallel for
for (i=0; i<m; i++)
{ y[i] = a * x[i] + y[i];}
```

- Loop indices are distributed among threads
- Parameters 'a, m, x' are read-only
- Parameter 'y' is shared
  - All threads update it, but each at different memory location
- Parameter 'i' is private, loop index!
  - Each thread has its own 'i' value and range
  - 'i' becomes undefined after "parallel for"

Example

```
#pragma omp parallel
#pragma omp for
for(i=0; i<imax; i++)
{ a[i] = b[i];}
#pragma omp for
for(i=0; i<imax; i++)
{ c[i] = a[2];}
#pragma omp end parallel
```

Data Environment

- 'j' is private by default.

```
#pragma omp parallel for
for (j=0; j<m; j++) {
for (i=0; i<m; i++) {
//… calculation
} // i-loop
} // j-loop
```

- What is the **problem**?
- 'i' also needs to be private!
  - This is to avoid race condition among the threads
- Solution:

```
#pragma omp parallel for private(i)
for (j=0; j<m; j++) {
for (i=0; i<m; i++) {
//… calculation
} // i-loop
} // j-loop
```

- Two more examples:

```
#pragma omp parallel for
ifirst = 5;

#pragma omp parallel for default(none) \
shared(ifirst,ilast,j) private(x)
for(i = 0; i < ilast; i++){
   x = 2*i;
   a[i] = ifirst + x;
}
```

```
#pragma omp parallel for default(none) \
private(i,j,sum) shared(n,m,a,b,c)
for (i=0; i<n; i++)
{
sum = 0;
for (j=0; j<m; j++)
sum += x[i][j] * y[j];
z[i] = sum;
}
```

Data Environment

- Private: there is no connection between the original variable and the private copies created
- Firstprivae is same as private but the variable's initial value is copied from the main one
- Lastprivate is same as private but the variable's last value is copied to the main one

```
int z=1; int x=10;
#pragma omp parallel for firstprivate(x) lastprivate(x)
for (i=0; i<n; i++) {
if (data[i]==x)
z=i;
}
```

- firstprivate(z) creates a private memory location for each thread
- lastprivate(i) saves the value of the last loop index

```
z=0;
#pragma omp parallel for firstprivate(z)
for(i = 0; i < ilast; i++){
   z = z + 1;
   h[i] = z;
}
```

```
#pragma omp parallel for lastprivate(i)
for(i = 0; i < ilast-1; i++){
   h[i] = a[i];
}
h(i) = a(1);
```

Single and Master constructs

- The 'single' region is executed by one thread. This is useful for shared variables initialization

```
#pragma omp single
{ a = 10; }
#pragma omp for
{ for (i=0; i<m; i++)
b[i] = a;
}
```

- Let's take the following example:

```
#include <stdio.h>
void work1() {}
void work2() {}
void single_example()
{
  #pragma omp parallel
  {
  #pragma omp single
  printf("Beginning work1.\n");
  work1();
  #pragma omp single
  printf("Finishing work1.\n");
  #pragma omp single nowait
  printf("Finished work1 and beginning work2.\n");
  work2();
  }
}
```

Single and Master constructs

- In the last example:

- One thread at a time can execute the single region
- A barrier is implied at the end of single block

- Other threads will skip it and stop at the barrier – end of single construct, till all threads in the team have reached the barrier

- nowait clause is added because other threads can proceed without waiting for the one executing the single region

- The master construct is only executed by the master thread. Other threads skip it. No synchronization is implied

Sections

- One thread execute each section. Each section is executed only once!
- If there are too many sections, some threads execute more than one section (in a round robin fashion)
- If there are few sections, some threads are idle
- We don't know in advance which thread will execute which section

```
#pragma omp sections
{
#pragma omp section
{ foo(); }
#pragma omp section
{ bar(); }
#pragma omp section
{ beer(); }
} // end of sections
```

```
#pragma omp parallel 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 sections
```

Input thread

Processing thread(s)

Output thread

Critical

- Only one threat at a time can enter the critical section
- Also, use #pragma omp atomic: faster than critical and protects only a single assignment
- Global data update can use atomic operations too.

- #pragma omp critical { block of codes }
- #pragma omp atomic { only one statement }
- #pragma omp barrier

```
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 <name>
   consume(B, RES);
   }
}
```

```
c = 0; f = 7;
#pragma omp parallel
{

#pragma omp for
for (i=0;i<20;i++)
{
if (b[i] == 0) {

#pragma omp critical
c ++;
}

a[i] = b[i]+f*(i+1);
}
} /*omp end parallel */
```

Reduction

- Different threads may simultaneously write to “sum”
- Parallel for will not parallelize correctly

```
Sum = 0;
for (i=0; i<maxi; i++) {
sum = sum + a[i];
}
```

- Use reduction!
- A private copy of each variable is created and initialized
- Copies are updated by threads
- Results of threads at the end of the loop are automatically summed (reduced)

```
Sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<maxi; i++)
{ sum = sum + a[i]; }
```

Reduction: Dot Product

- The private(sum) will not work. Sum is not private. Though, accessing sum atomically is expensive!

```
float dot_product (float *a, float * b, int N)
{
   float sum = 0.0
   #pragma omp parallel for private(sum)
   for(int i=0; i<N; i++)
   sum += a[i]*b[i];
}
```

- It should be as follows:

```
float dot_product (float *a, float * b, int N)
{
   float sum = 0.0;
   #pragma omp parallel for reduction(+:sum)
   for(int i=0; i<N; i++)
   sum += a[i]*b[i];
}
```

- This is a private copy of sum in each thread.
- Copies are then added up

Reduction: Numerical Integration

- Numerical integration (sum of rectangles)
- F(x) = 4/(1+x*x)

```
static long num_rects = 100; double step;
int main ()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_rects;
   for (i=0;i< num_rects; i++){
   x = (i+0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

- Shared variable(s)? num_rects, step
- Private variable(s)? x, i
- Reduced variable(s)? sum

```
static long num_rects = 100; double step;
int main ()
{
   int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_rects;

   #pragma omp parallel for private(x) reduction(+:sum)
   for (i=0;i< num_rects; i++){
   x = (i+0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Reduction: Max, Min

```
#include <math.h>
void reduction1 (float *x, int *y, int n)
{
   int i, b, c;
   float a, d;
   a = 0.0; b = 0;
   c = y[0]; d = x[0];
   #pragma omp parallel for private(i) shared(x, y, n) reduction(+:a) reduction(^:b) \
   reduction(min:c) reduction(max:d)

   for (i=0; i<n; i++) {
   a += x[i];
   b ^= y[i];
   if (c > y[i]) c = y[i];
   d = fmaxf(d,x[i]);
}
```

Reduction

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
  for(int i=0; i<N; i++)
     { sum += a[i] * b[i]; }
  return sum;
}
```

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++)
          { sum += a[i] * b[i]; }
  return sum;
}
```

- Still , there is a problem!! The shared variable 'sum' is NOT protected from DATA RACES!
- Solutions are as follows:

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;

  #pragma omp parallel for shared(sum)
  for(int i=0; i<N; i++) {

      #pragma omp critical
      sum += a[i] * b[i];
  }
   return sum;
}
```

OR

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;

   #pragma omp parallel for reduction(+:sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

Nowait

- The case of having multiple independent loops inside a parallel region?
- Solution? nowait clause
- This is to avoid the implied barrier at the end of loop construct
- It's used to minimize synchronization – threads don't wait or synchronize together at the end of the nowait construct

```
#include <math.h>
void nowait_example(int n, int m, float *a, float *b, float *y, float *z)
{
   int i;
   #pragma omp parallel
   {

   #pragma omp for nowait
   for (i=1; i<n; i++)
   b[i] = (a[i] + a[i-1])/2;

   #pragma omp for nowait
   for (i=0; i<m; i++)
   y[i] = sqrt(z[i]);
   }
}
```

Example: The Cumulative Sum

- Compute the cumulative sum:

```
sum = 0;
for (i=0; i<m; i++)
sum += a[i];
```

```
sum = 0;
#pragma omp parallel for
for (i=0; i<m; i++)
sum += a[i];
```

- This is wrong!! Why? RACE CONDITION
- We need one thread to execute the sum+ region at a time! MUTUAL EXCLUSIVE
- Use critical! Also, reduction and atomic are used to avoid race condition

```
sum = 0;
#pragma omp parallel shared(n,a,sum)
private(sum_local)
{
sum_local = 0;
#pragma omp for
for (i=0; i<m; i++)
sum_local += a[i]; // per-thread local sum
#pragma omp critical
{ sum += sum_local; } // global sum
}
```

```
sum = 0;
#pragma omp parallel for shared(n,a,sum) \
private(sum_local) reduction(+:sum)
{
for (i=0; i<m; i++)
sum += a[i];
}
```

More Examples

- Initialization of ‘a’ should be synchronized with the reduction of ‘a’
- As such, ‘a’ must be initialized before any update of ‘a’ in the for loop

- Though, this could be *also* achieved by:
  - Initializing ‘a’ before the of parallel region,
  - Enclosing a=0 in a single directive,
  - Or adding an explicit barrier after a=0

```
#include <stdio.h>
int main (void)
{
   int a, i;

   #pragma omp parallel shared(a) private(i)
   {

   #pragma omp master
   a = 0;
   // To avoid race conditions, add a barrier here

   #pragma omp for reduction(+:a)
   for (i = 0; i < 10; i++) {
   a += i;
   }

   #pragma omp single
   printf ("Sum is %d\n", a);
}
```

More Examples

```
void* work(float* c, int N) {
  float x, y; int i;
  for(i=0; i<N; i++)
     { x = a[i]; y = b[i];
       c[i] = x + y; }
}
```

```
void* work(float* c, int N) {
  float x, y; int i;
  #pragma omp parallel for private(x,y)
  for(i=0; i<N; i++)
     { x = a[i]; y = b[i];
       c[i] = x + y; }
}
```

- Also, use firstprivate(x) if initialization is necessary

More Examples

- Example: Which loop to execute in parallel?

```
main() {
int i, j, k; float **a, **b;
for(k=0; k<n;k++) \\ Loop carried dependence
   for(i=0;i<n;i++) \\ Can be executed in parallel
      for(j=0; j<n;j++) \\ Can be executed in parallel
         a[i][j] = min(a[i][j], a[i][k]+a[k][j]);
}
```

- How to maximize the amount of work/gain for each fork-join?
  - Parallelize middle loop!

More Examples

```
main() {
int i, j, k; float **a, **b;
for(k=0; k<n;k++) \\ Loop carried dependence
   #pragma omp parallel for
   for(i=0;i<n;i++) \\ Can be executed in parallel
      for(j=0; j<n;j++) \\ Can be executed in parallel
         a[i][j] = min(a[i][j], a[i][k]+a[k][j]);
}
```

- Still we have a problem? 'j' is shared!

```
main() {
int i, j, k; float **a, **b;
for(k=0; k<n;k++) \\ Loop carried dependence
   #pragma omp parallel for private(j)
   for(i=0;i<n;i++) \\ Can be executed in parallel
      for(j=0; j<n;j++) \\ Can be executed in parallel
         a[i][j] = min(a[i][j], a[i][k]+a[k][j]);
}
```

More Examples

```
#include <stdio.h>
extern float average(float,float,float);
void master_example( float* x, float* xold, int n, float tol )
{
   int c, i, toobig; float error, y; c = 0;
   #pragma omp parallel {
   do {

   #pragma omp for private(i)
   for( i = 1; i < n-1; ++i )
   { xold[i] = x[i]; }

   #pragma omp single
   { toobig = 0; }

   #pragma omp for private(i,y,error) reduction(+:toobig)
   for( i = 1; i < n-1; ++i )
   {
   y = x[i];
   x[i] = average( xold[i-1], x[i], xold[i+1] );
   error = y - x[i];
   if( error > tol || error < -tol ) ++toobig;
   }

   #pragma omp master
   { ++c;
   printf( "iteration %d, toobig=%d\n", c, toobig );
   }
   }
   while( toobig > 0 );
   }
}
```

More Examples

```
#include <omp.h>
void main ()
{
   int i;
   double z, func(), res=0.0;
   #pragma omp parallel for reduction(+:res) private(z)
   for (i=0; i< 100; i++){
   z = func(i);
   res = res + z;
   }
}
```

```
void mxv_row(int m, int n, double *a, double *b, double *c)
{
   int i, j;
   double sum;

   #pragma omp parallel for default(none) private(i,j,sum) shared(m,n,a,b,c)
   for (i=0; i<m; i++)
   {
   sum = 0.0;
   for (j=0; j<n; j++)
   sum += b[i*n+j]*c[j];
   a[i] = sum;
   }
}
```

More Examples

```
sum = 0;
#pragma omp parallel private (lsum)
{
   lsum = 0;
   #pragma omp for
   for (i=0; i<N; i++) {
      lsum = lsum + A[i];
   }
   #pragma omp critical
   { sum += lsum; }
}
```

Only one thread at a time execute the critical section

OR

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

sum is shared variable!

More Examples

```
#include <limits.h>
#include <math.h>
void reduction2(float *x, int *y, int n)
{
   int i, b, b_p, c, c_p;
   float a, a_p, d, d_p;
   a = 0.0f; b = 0; c = y[0]; d = x[0];

   #pragma omp parallel shared(a, b, c, d, x, y, n) private(a_p, b_p, c_p, d_p)
   {
   a_p = 0.0f; b_p = 0; c_p = INT_MAX; d_p = -HUGE_VALF;

   #pragma omp for private(i)
   for (i=0; i<n; i++) {
   a_p += x[i];
   b_p ^= y[i];
   if (c_p > y[i]) c_p = y[i];
   d_p = fmaxf(d_p,x[i]);
   }

   #pragma omp critical
   {
   a += a_p;
   b ^= b_p;
   if( c > c_p ) c = c_p;
   d = fmaxf(d,d_p);
   }
}
```

More Examples

```
void mxv_row (int m, int n, double *a, double *b, double *c)
{
   int i, j;
   double sum;
   #pragma omp parallel for default(none) private(i,j,sum) shared(m,n,a,b,c)

   for (i=0; i<m; i++)
   {
      sum = 0;
      for (j=0; j<n; j++)
      sum += b[i][j]*c[j];
      a[i] = sum;
   }
}
```

References

- OpenMP forum, http://www.openmp.org/forum/
- OpenMP tutorial, https://computing.llnl.gov/tutorials/openMP/
- OpenMP Org, http://openmp.org/wp/
- Guide into OpenMP: Easy multithreading programming for C++ http://bisqwit.iki.fi/story/howto/openmp/
- C++ – OpenMP – Examples of basic parallel programming, http://berenger.eu/blog/