COMS3008A: Parallel Computing
Introduction to OpenMP: Part I

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

2021-9-9

Contents

WITS UNIVERSITY

Outline

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Loop construct
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct

Outline

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Loop construct
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct

WITS UNIVERSITY

Worksharing construct

- A parallel construct by itself creates an SPMD program, i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team? Worksharing.
- Worksharing
  - **for** construct or loop construct: Splits up a loop iterations among the threads in a team.
  - `sections/section` constructs
  - `task` construct
  - `single` construct
- The following construct is also discussed as synchronization construct.
  - `master` construct

WITS UNIVERSITY

Outline

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Loop construct
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct

WITS UNIVERSITY

Loop construct

Loop construct: **#pragma** `omp` **for**

Example 1

Given arrays $A[N]$ and $B[N]$. Find $A[i] = A[i] + B[i], 0 \leq i \leq N - 1$.

```
......
//Serial
for(i=0; i< N; i++)
  a[i]=a[i]+b[i];
```

```
......
//Using loop construct
#pragma omp parallel
#pragma omp for
{
  for(i=0; i<N; i++)
    a[i] = a[i] + b[i];
}
```

```
//Using parallel construct
#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];
}
```

WITS UNIVERSITY

Loop construct cont.

- The **schedule** clause specifies how the iterations of the loop are assigned to the threads in a team.
- The syntax is: **#pragma** `omp` **for** `schedule(kind [, chunk])`.
- Schedule kinds:
  - **schedule(static [,chunk])**: Deals out blocks of iterations of size `chunk` to each thread in a round robin fashion.
    - The iterations can be assigned to the threads before the loop is executed.
  - **schedule(dynamic [,chunk])**: Each thread grabs `chunk` size of iterations off a queue until all iterations have been handled. The default is 1.
    - The iterations are assigned while the loop is executing

Schedule clause

- **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. The default is 1.
- **schedule(runtime)**: Schedule and chunk size taken from the OMP_SCHEDULE environment variable.
  - For example, `export OMP_SCHEDULE="static,1"`
- **schedule(auto)**: The selection of the schedule is determined by the implementation.

WITS UNIVERSITY

More on schedule

- Most OpenMP implementations use a roughly block partition.
- There is some overhead associated with schedule.
- The overhead for `dynamic` is greater than `static`, and the overhead for `guided` is the greatest.
- If each iteration of a loop requires roughly the same amount of computation, then it is likely that the default distribution will give the best performance.
- If the cost of the iterations decreases linearly as the loop executes, then a static schedule with small chunk size will probably give the best performance.
- If the cost of each iteration can not be determined in advance, then `schedule(runtime)` can be used.

WITS UNIVERSITY

Ordered construct/clause

- `ordered` construct: This is a synchronization construct. It allows one to execute a structured block within a parallel loop in sequential order.
- An `ordered` clause has to be added to the parallel region in which the `ordered` construct appears; it informs the compiler that the construct occurs.

```c
#pragma omp parallel private(i, TID) shared(a) ordered
#pragma omp for
for(i=0; i<n; i++) {
  TID = omp_get_thread_num();
  printf("Thread %d updates %dth item in the array\n", TID, i);
  a[i] += i;
  #pragma omp ordered
  printf("Thread %d prints %dth value of the array\n", TID, i);
}
```

Loop construct cont.

Basic approach to parallelize a loop:

- 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 directives and test.

Loop carried dependency

- The computation of one iteration depends on the results of one or more previous iterations.
- The program could compile without errors. However, the result can be incorrect or unpredictable.
- A loop with loop-carried dependency cannot, in general, be correctly parallelized by OpenMP, unless the loop-carried dependency is removed.

```
fibo[0] = fibo[1] = 1;
#pragma omp parallel num_threads(thread_count)
#pragma omp for
for(i = 2; i < n; i++)
  fibo[i] = fibo[i-1] + fibo[i-2];
```

Loop carried dependency cont.

Example 2

Removing a loop carried dependency.

```c
//Loop dependency
int i, j, A[MAX];
j=5;
for(i=0; i<MAX; i++){
  j+=2;
  A[i]=big(j);
}
//Removing loop dependency
int i, A[MAX];
#pragma omp parallel
  #pragma omp for
  for(i=0; i<MAX; i++){
    int j=5+2*(i+1);
    A[i]=big(j);
}
```

Loop carried dependency cont.

- Loop carried dependency: Dependencies between instructions in different iterations of a loop;
- What are the dependencies in the following loop?

```
for(i=0; i<N; i++) {
  B[i]=tmp;
  A[i+1]=B[i+1];
  tmp=A[i];
}
```

Loop carried dependency cont.

It helps to unroll the loop to see the dependencies.

```
    i=0:
      B[0]=tmp;
      A[1]=B[1];
      tmp=A[0];

    i=1:
      B[1]=tmp;
      A[2]=B[2];
      tmp=A[1];

    i=2:
      B[2]=tmp;
      A[3]=B[3];
      tmp=A[2];
    ......
```

WITS UNIVERSITY

Reduction

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

We are aggregating multiple values into a single value—**reduction**. Reduction operation is supported in most parallel programming environments.

- OpenMP reduction clause: *reduction(op:list)*.
- Inside a parallel for worksharing construct
  - A local copy of each list variable is made and initialized depending on the operation specified by the operator “op”.
  - Each thread updates its own local copy
  - Local copies are aggregated into a single value.

WITS UNIVERSITY

Reduction cont.

```
......
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;
......
```

- Associative operands that can be used with reduction (for C/C++) and their common initial values.

Op	Initial value	Op	Initial value
+	0	&	∼0
*	1	|	0
-	0	^	0
min	Large number (+)	&&	1
max	Most neg. number	||	0

WITS UNIVERSITY

Collapse clause

The `collapse` Clause

```
void work(int a, int j, int k);
void main() {
  int j, k, a;
  int m = 2, n = 5;
  #pragma omp parallel num_threads(4)
  {
    #pragma omp for private(i,j,k)
    for (k=0; k<m; k++)
      for (j=0; j<n; j++) {
        printf("%d %d %d\n", i, k, j);
        work(a,j,k);
      }
  }
}
```

- The iterations of the `k` and `j` loops are collapsed into one loop, and that loop is then divided among the threads in the current team.

Collapse clause cont.

```c
void work(int a, int j, int k);
void main() {
  int t, a;
  int m = 2, n = 5;
  #pragma omp parallel num_threads(4)
  {
    #pragma omp for private(j,k) schedule(static,2)
    for(t=0; t<10; t++) {
      printf("%d %d %d\n", omp_get_thread_num(), (t/n)%m, t%n);
      /* end ordered */
      work(a,t%n,(t/n)%m);
    }
  }
}
```

Collapse clause cont.

Example 3

`collapse` clause example

```c
void work(int a, int j, int k);
void main() {
  int j, k, a;
  int m = 2, n = 5;
  #pragma omp parallel num_threads(2)
  {
    #pragma omp for collapse(2) ordered private(j,k)
        schedule(static,2)
    for (k=0; k<m; k++)
      for (j=0; j<n; j++) {
        #pragma omp ordered
        printf("%d %d %d\n", omp_get_thread_num(), k,
             j);
        /* end ordered */
        work(a,j,k);
      }
  }
}
```

Clauses supported by the loop construct

- **private**
- `firstprivate`
- `lastprivate`
- `reduction`
- `schedule`
- `ordered`
- `nowait`
- `collapse`

Examples — lastprivate clause

Example 4

`lastprivate` clause example

```
void lastpriv (int n, float *a, float *b) {
  int i, a, n = 5;
  ......
  #pragma omp parallel private(i) lastprivate(a)
  #pragma omp for
  for (i=0; i<n; i++) {
    a = i+1;
    printf("Thread %d has a value of a = %d for i = %d\
        n", omp_get_thread_num(),a,i);
  } /*-- End of parallel for --*/
  printf("Value of 'a' after parallel for: a=%d\n",a);
}
```

Examples — lastprivate clause cont.

Example 5

`lastprivate` clause example

```
void sq2(int n, double *lastterm) {
  double x; int i;
  #pragma omp parallel for lastprivate(x)
  for(int i=0; i<1000; i++) {
    x=a[i]*a[i]+b[i]*b[i];
    b[i]=sqrt(x);
  }
  /*x has the value it held for the last sequential
      iteration, i.e., for i=(1000-1)*/
  *lastterm = x;
}
```

Examples — nowait clause

Example 6

`nowait` clause example

```
#include <math.h>
void nowait_example2(int n, float *a, float *b, float *
    c, float *y, float *z) {
  int i;
  #pragma omp parallel
  {
    #pragma omp for schedule(static)
    for (i=0; i<n; i++) {
      c[i] = (a[i] + b[i]) / 2.0f;
    }/*implicit barrier*/
    #pragma omp for schedule(static) nowait
    for (i=0; i<n; i++) {
      z[i] = sqrtf(c[i]);
    }/*no implicit barrier due to nowait clause*/
    #pragma omp for schedule(static) nowait
    for (i=1; i<=n; i++)
      y[i] = z[i-1] + a[i];
    /*no implicit barrier due to nowait clause*/
  }/*implicit barrier at the end of a parallel region,
     cannot be removed*/
}
```

More on for construct

- OpenMP parallelizes `for` loops that are in canonical form.
- `for` loop must not contain statements that allow the loop to be exited prematurely, such as `break`, `return`, or `exit` statements. The `continue` statement is allowed.
- Loops in canonical form take one of the following forms.

```
                                         index++
                                         ++index
                     index <  end        index--
                     index <= end        --index
for( index=start;    index >= end;       index += incr     )
                     index >  end        index -= incr
                                         index=index+incr
                                         index=incr+index
                                         index=index-incr
```

WITS UNIVERSITY

Outline

Combined parallel worksharing construct

Combined parallel worksharing constructs are shortcuts that can be used when a parallel region comprises precisely one worksharing construct.

```
#pragma omp parallel
  #pragma omp for
    for-loop
```

```
//combined for version
#pragma omp parallel for
  for-loop
```

Outline

WITS UNIVERSITY

Single worksharing construct

- The `single` construct denotes a block of code that is executed by only one thread.
- Syntax:

```
    #pragma omp single [clause[[,] clause]...]
    structured block
```

- clauses: `private`, `firstprivate`, `copyprivate`, `nowait`
- A barrier is implied at the end of the *single* block, unless a `nowait` clause is specified.
- This construct is ideally suited for I/O or initialization.

Single worksharing construct cont.

Example 7

`single` construct example

```
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();
  }
}
```

Copyprivate clause

- `copyprivate` clause is used with `single` construct only.
- It provides a mechanism to broadcast the value of a private variable from one thread to the rest of the team.

Example 8

`copyprivate` clause example

```
int TID;
float rate=1.2;
omp_set_num_threads(4);
#pragma omp parallel private(rate,TID)
{
  TID = omp_get_thread_num();
  #pragma omp single copyprivate(rate)
  {
    rate = rand()*1.0/RAND_MAX;
  }
  printf("Value for variable rate: %f by thread %d\n",
       rate, TID);
}
```

Outline

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Loop construct
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct

WITS UNIVERSITY

Master construct

`master` construct:

- The `master` construct specifies a structured block that is executed by the master thread of the team.
- There is no implied barrier either on entry to, or exit from, the master construct.

The number of threads active

The `omp_set_dynamic` routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions.

- omp_set_dynamic() – A call to this function with nonzero argument allows OpenMP to choose any number of threads between 1 and the set number of threads.

Example 9

```
omp_set_dynamic(1);
#pragma omp parallel num_threads(8)
```

allows the OpenMP implementation to choose any number of threads between 1 and 8.

The number of threads active cont.

Example 10

```
omp_set_dynamic(0);
#pragma omp parallel num_threads(8)
```

only allows the OpenMP implementation to choose 8 threads. The action in this case is implementation dependent.

- omp_get_dynamic() – You can determine the default setting by calling this function.

References

- Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation), by Barbara Chapman, Gabriele Jost and Ruud van der Pas. The MIT Press, 2007.
- Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD, by Ruud van der Pas, Eric Stozer, and Christian Terboven. The MIT Press, 2017.
- `https://hpc.llnl.gov/tuts/openMP/`