

COMS3008A: Parallel Computing

Introduction to OpenMP: Part II

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

March 20, 2025

1 OpenMP Core Features

- Worksharing in OpenMP
- Loop construct
- Combined parallel worksharing constructs
- Single worksharing construct
- Master construct

1 OpenMP Core Features

- Worksharing in OpenMP
- Loop construct
- Combined parallel worksharing constructs
- Single worksharing construct
- Master construct

1 OpenMP Core Features

- Worksharing in OpenMP
 - Loop construct
 - Combined parallel worksharing constructs
 - Single worksharing construct
 - Master construct

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

1 OpenMP Core Features

- Worksharing in OpenMP
- **Loop construct**
- Combined parallel worksharing constructs
- Single worksharing construct
- Master construct

Loop construct

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

Example 1

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

```
1 .....  
2 //Serial  
3 for (i=0; i < N; i++)  
4   a[i]=a[i]+b[i];
```

```
1 .....  
2 //Using loop construct  
3 #pragma omp parallel  
4 #pragma omp for  
5 {  
6   for (i=0; i < N; i++)  
7     a[i] = a[i] + b[i];  
8 }
```

```
1 //Using parallel construct  
2 #pragma omp parallel  
3 {  
4   int id, i, nthrds, istart,  
5     iend;  
6   id = omp_get_thread_num();  
7   nthrds =  
8     omp_get_num_threads();  
9   istart = id * N/nthrds;  
10  iend = (id+1) * N/nthrds;  
11  if (id==nthrds-1)  
12    iend = N;  
13  for (i=istart; i<iend; i++)  
14    a[i] = a[i] + b[i];  
15 }
```



Clauses supported by the loop construct

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

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.

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.

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.

```
1 #pragma omp parallel private (i, TID) shared(a)
2 #pragma omp for ordered
3 for (i=0; i<n; i++) {
4     TID = omp_get_thread_num();
5     printf("Thread %d updates %dth item in the array\n",
6           TID, i);
7     a[i] += i;
8     #pragma omp ordered
9     printf("Thread %d prints %dth value of the array\n",
10           TID, i);
11 }
```

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.

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

Loop carried dependency cont.

Example 2

Removing a loop carried dependency.

```
1 //Loop dependency
2 int i, j, A[MAX];
3 j=5;
4 for (i=0; i<MAX; i++){
5     j+=2;
6     A[i]=big(j);
7 }
8 //Removing loop dependency
9 int i, A[MAX];
10 #pragma omp parallel
11     #pragma omp for
12     for (i=0; i<MAX; i++){
13         int j=5+2*(i+1);
14         A[i]=big(j);
15 }
```

Loop carried dependency cont.

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

```
1  for (i=0; i<N; i++) {  
2      B[i]=tmp;  
3      A[i+1]=B[i+1];  
4      tmp=A[i];  
5  }
```


Loop carried dependency cont.

It helps to unroll the loop to see the dependencies.

```
1      i=0 :  
2          B[0]=tmp;  
3          A[1]=B[1];  
4          tmp=A[0];  
  
6      i=1 :  
7          B[1]=tmp;  
8          A[2]=B[2];  
9          tmp=A[1];  
  
11     i=2 :  
12         B[2]=tmp;  
13         A[3]=B[3];  
14         tmp=A[2];  
15     . . . . .
```

Reduction

```
1 .....  
2 double ave=0.0, A[MAX];  
3 int i;  
4 for (i=0; i<MAX; i++) {  
5     ave+=A[i];  
6 }  
7 ave = ave/MAX;  
8 .....
```

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.

Reduction cont.

```
1 .....  
2 double ave=0.0, A[MAX];  
3 int i;  
4 #pragma omp parallel for  
   reduction(+:ave)  
5     for (i=0;i<MAX;i++) {  
6         ave+=A[i];  
7     }  
8 ave = ave/MAX;  
9 .....
```

- 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

Collapse clause

The collapse Clause

```
1 void work(int a, int j, int k);
2 void main() {
3     int j, k, a;
4     int m = 2, n = 5;
5     #pragma omp parallel num_threads(4)
6     {
7         #pragma omp for private(j,k)
8         for (k=0; k<m; k++)
9             for (j=0; j<n; j++) {
10                 printf("%d %d\n", k, j);
11                 work(a, j, k);
12             }
13     }
14 }
```

- 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.

```
1 void work(int a, int j, int k);
2 void main() {
3     int t, a;
4     int m = 2, n = 5;
5     #pragma omp parallel num_threads(4)
6     {
7         #pragma omp for private(j,k) schedule(static,2)
8         for(t=0; t<10; t++) {
9             printf("%d %d %d\n", omp_get_thread_num(), (t/n)%
10                 m, t%n);
11             work(a,t%n, (t/n)%m);
12         }
13     }
```

Collapse clause cont.

Example 3 (collapse clause example)

```
1 void work(int a, int j, int k);
2 void main() {
3     int j, k, a;
4     int m = 2, n = 5;
5     #pragma omp parallel num_threads(2)
6     {
7         #pragma omp for collapse(2) ordered private(j,k)
8         schedule(static,2)
9         for (k=0; k<m; k++)
10            for (j=0; j<n; j++) {
11                #pragma omp ordered
12                printf("%d %d %d\n", omp_get_thread_num(), k,
13                    j);
14                /* end ordered */
15                work(a,j,k);
16            }
17    }
```

Examples — lastprivate clause

Example 4 (lastprivate clause example)

```
1 void lastpriv (int n, float *a, float *b) {
2     int i, a, n = 5;
3     .....
4     #pragma omp parallel private(i) lastprivate(a)
5     #pragma omp for
6     for (i=0; i<n; i++) {
7         a = i+1;
8         printf("Thread %d has a value of a = %d for i = %d\
          n", omp_get_thread_num(),a,i);
9     } /*-- End of parallel for --*/
10    printf("Value of 'a' after parallel for: a=%d\n",a);
11 }
```

Examples — lastprivate clause cont.

Example 5 (lastprivate clause example)

```
1 void sq2(int n, double *lastterm) {  
2     double x; int i;  
3     #pragma omp parallel for lastprivate(x)  
4     for(int i=0; i<1000; i++) {  
5         x=a[i]*a[i]+b[i]*b[i];  
6         b[i]=sqrt(x);  
7     }  
8     /*x has the value it held for the last sequential  
9        iteration, i.e., for i=(1000-1)*/  
10    *lastterm = x;  
11 }
```


Examples — nowait clause

Example 6 (nowait clause example)

```
1  #include <math.h>
2  void nowait_example2(int n, float *a, float *b, float *
   c, float *y, float *z) {
3      int i;
4      #pragma omp parallel
5      {
6          #pragma omp for schedule(static)
7          for (i=0; i<n; i++) {
8              c[i] = (a[i] + b[i]) / 2.0f;
9          } /*implicit barrier*/
10         #pragma omp for schedule(static) nowait
11         for (i=0; i<n; i++) {
12             z[i] = sqrtf(c[i]);
13         } /*no implicit barrier due to nowait clause*/
14         #pragma omp for schedule(static) nowait
15         for (i=1; i<=n; i++)
16             y[i] = z[i-1] + a[i];
17         /*no implicit barrier due to nowait clause*/
18     } /*implicit barrier at the end of a parallel region,
       cannot be removed*/
19 }
```

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
```

1 OpenMP Core Features

- Worksharing in OpenMP
- Loop construct
- **Combined parallel worksharing constructs**
- Single worksharing construct
- Master construct

Combined parallel worksharing construct

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

```
1 #pragma omp parallel
2   #pragma omp for
3     for-loop
```

```
1 //combined for version
2 #pragma omp parallel for
3   for-loop
```

1 OpenMP Core Features

- Worksharing in OpenMP
- Loop construct
- Combined parallel worksharing constructs
- **Single worksharing construct**
- Master construct

Single worksharing construct

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

```
1  #pragma omp single [clause[[,] clause]...]  
2  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)

```
1 void work1() {}
2 void work2() {}
3 void single_example() {
4     #pragma omp parallel
5     {
6         #pragma omp single
7         printf("Beginning work1.\n");
8         work1();
9         #pragma omp single
10        printf("Finishing work1.\n");
11        #pragma omp single nowait
12        printf("Finished work1 and beginning work2.\n");
13        work2();
14    }
15 }
```

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)

```
1 int TID;
2 float rate=1.2;
3 omp_set_num_threads(4);
4 #pragma omp parallel private (rate,TID)
5 {
6     TID = omp_get_thread_num();
7     #pragma omp single copyprivate(rate)
8     {
9         rate = rand()*1.0/RAND_MAX;
10    }
11    printf("Value for variable rate: %f by thread %d\n",
12           rate, TID);
13 }
```


1 OpenMP Core Features

- Worksharing in OpenMP
- Loop construct
- Combined parallel worksharing constructs
- Single worksharing construct
- **Master construct**

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.

- 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.
- <https://hpc-tutorials.llnl.gov/openmp/>