COMS4040A & COMS7045A: High Performance Computing & Scientific Data Management Introduction to OpenMP: Part II

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

March 10, 2025

WITS UNIVERSITY

Contents

WITS UNIVERSITY

Outline

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct
   - Sections/Section Construct
   - Task worksharing construct

Outline

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct
   - Sections/Section Construct
   - Task worksharing construct

More on schedule cont.

- 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.

```
#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 1

Removing a loop carried dependency.

```
//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-1; 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];
    ......
```

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.

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

```
void work(int a, int j, int k);
int main() {
  int j, k, a[10];
  int m = 2, n = 5;
  ......
  #pragma omp parallel num_threads(4)
  {
    #pragma omp for private(j,k)
    for (k=0; k<m; k++)
      for (j=0; j<n; j++) {
        a[k*n+j] = k*n+j;
        printf("%d %d\n", k, j);
        work(a,j,k);
      }
  }
}
```

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

Collapse clause cont.

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

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

Collapse clause cont.

Example 2 (collapse clause example)

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

WITS UNIVERSITY

Collapse clause cont.

Example 3 (`collapse & ordered` clause example)

```c
void work(int a, int j, int k);
int main() {
  int j, k, a;
  int m = 2, n = 5;
  #pragma omp parallel num_threads(2) private(j,k)
  {
    #pragma omp for collapse(2) schedule(static,2) ordered
    for (k=0; k<m; k++)
      for (j=0; j<n; j++) {
        a[k*n+j] = k*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)

```c
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;
}
```

WITS UNIVERSITY

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
```

Outline

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct
   - Sections/Section Construct
   - Task worksharing construct

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

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct
   - Sections/Section Construct
   - Task worksharing construct

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
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct
   - Sections/Section Construct
   - Task worksharing 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.

WITS UNIVERSITY

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.

Outline

1. OpenMP Core Features
   - Worksharing in OpenMP
   - Combined parallel worksharing constructs
   - Single worksharing construct
   - Master construct
   - Sections/Section Construct
   - Task worksharing construct

Sections/Section Construct

- `sections` directive enables specification of **task parallelism**
- The `sections` worksharing construct gives a different structured block to each thread.
- Syntax:

```
#pragma omp sections [clause[[,] clause]...]
{
  [#pragma omp section]
  structured block
  [#pragma omp section]
  structured block
  ...
}
```

- clauses: `private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`
- Each section must be a structured block of code that is independent of other sections.
- There is an implicit barrier at the end of a sections construct unless a `nowait` clause is specified.

WITS UNIVERSITY

Examples — firstprivate

```
#include <omp.h>
#include <stdio.h>
#define NT 4
int main( ) {
  int section_count = 0;
  omp_set_dynamic(0);
  omp_set_num_threads(NT);
  #pragma omp parallel
  #pragma omp sections firstprivate( section_count )
  {
    #pragma omp section
    {
      section_count++;
      printf( "section_count %d\n", section_count );
    }
    #pragma omp section
    {
      section_count++;
      printf( "section_count %d\n", section_count );
    }
  }
  return 0;
}
```

Example – Parallel quicksort

Example 11

Parallelize the sequential *quicksort* program (`qsort_serial.c`) using OpenMP `sections/section` construct.

```
q_sort(left, right, data) {
  if(left < right){
    q = partition(left, right, data);
    q_sort(left, q-1, data);
    q_sort(q+1, right, data);
  }
}
partition(left, right, float *data) {
  x = data[right];
  i = left-1;
  for(j=left; j<right; j++) {
    if(data[j] <= x){
      i++;
      swap(data, i, j);
    }
  }
  swap(data, i+1, right);
  return i+1;
}
```

Outline

Task worksharing construct

- Tasks are independent units of work.
- Threads are assigned to perform the work of each task.
  - Tasks may be deferred
  - Tasks may be executed immediately
- The runtime system decides which of the above
- A *task* is composed of:
  - Code to execute
  - A data environment
  - Internal control variables, such as
    - `OMP_NESTED` – `TRUE` or `FALSE`, controls whether nested parallelism is enabled.
    - `OMP_DYNAMIC` – `TRUE` or `FALSE`, Enables or disables dynamic adjustment of the number of threads in parallel regions. Library functions `omp_set_dynamic()`, – sets `OMP_DYNAMIC` to be true or false, and `omp_get_dynamic()` – returns how `OMP_DYNAMIC` is set.
    - `OMP_NUM_THREADS` – library function `omp_set_num_threads()` can set this env variable.
    - `OMP_SCHEDULE` – Sets the default loop scheduling policy for `runtime` schedule clauses.

WITS
UNIVERSITY

Task construct cont.

Task Construct Syntax

#pragma omp task [clause[[,] clause]...]
    structured block

where clause can be

- if(*expr*): if *expr*=FALSE, then the task is immediately executed.
- shared
- private
- firstprivate
- default(shared|none)
- untied
- final(expr)

Task construct cont.

- `tied/untied`: Upon resuming a suspended task region, a `tied` task must be executed by the same thread again. With an `untied` task, there is no such restriction and any thread in the team can resume execution of the suspended task.
- **if** `(expr)` clause - If `expr` is evaluated to false, the task is undeferred and executed immediately by the thread that was creating the task.
- `final(expr)` clause - For recursive and nested applications, it stops task generations at a certain depth where we have enough tasks (or parallelism).

Task construct cont.

- Two activities: *packaging* and *execution*
  - Each encountering thread packages a new instance of task
  - Some thread in the team executes the task at some time later or immediately.
- Task barrier: The **taskwait** directive:

Task construct example

Example 12

```
......
#pragma omp parallel
{
  #pragma omp single private(p)
  {
    p=list_head;
    while(p){
      #pragma omp task
      processwork(p);
      p=p->next;
    }
  }
}
```

Task construct cont.

When tasks are guaranteed to be completed?

- At thread or task barriers
- At the directive: `#pragma omp barrier`
- At the directive: `#pragma omp taskwait`

Example 13

```
#pragma omp parallel
{
  #pragma omp task
  foo();
  #pragma omp barrier
  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
```

Task construct example

Example 14 (Understanding Task Construct)

```c
int main(int argc, char *argv[]){
  #pragma omp parallel num_threads(2)
  {
    printf("A ");
    printf("soccer ");
    printf("match ");
  }
  printf("\n");
  return 0;
}
```

WITS UNIVERSITY

Task construct example

Example 15 (Understanding Task Construct)

```
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      printf("soccer ");
      printf("match ");
    }
  }
  printf("\n");
  return 0;
}
```

Task construct example

Example 16 (Understanding Task Construct)

```c
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      printf("soccer ");
      #pragma omp task
      printf("match ");
    }
  }
  printf("\n");
  return 0;
}
```

WITS UNIVERSITY

Task construct example

Example 17 (Understanding Task Construct)

```c
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      printf("soccer ");
      #pragma omp task
      printf("match ");
      printf("is fun to watch ");
    }
  }
  printf("\n");
  return 0;
}
```

Task construct example

Example 18 (Understanding Task Construct)

```
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      printf("soccer ");
      #pragma omp task
      printf("match ");
      #pragma omp taskwait
      printf("is fun to watch ");
    }
  }
  printf("\n");
  return 0;
}
```

Task construct example

Example 19 (Tree traversal using task)

```
void traverse(node *p){
  if(p->left)
    #pragma omp task
    traverse(p->left);
  if(p->right)
    #pragma omp task
    traverse(p->right);
  process(p->data);
}
```

WITS
UNIVERSITY

Task construct example

Example 20 (Tree traversal using task)

```
    void traverse(node *p){
      if (p->left)
        #pragma omp task
        traverse(p->left)
      if (p->right)
        #pragma omp task
        traverse(p->right)
      #pragma omp taskwait
      process(p->data);
    }
```

Task construct example

Example 21

Write an OpenMP parallel program for computing the nth Fibonacci number. Compare the performance of the parallel implementation to the sequential one.

Task construct cont.

Task switching: *untied*:

```
#define ONEBILLION 1000000000L
#pragma omp parallel
{
  #pragma omp single
  {
    for(i=0; i<ONEBILLION; i++)
      #pragma omp task
      process(item[i]);
  }
  ......
  /* Untied task: any other thread is eligible to resume
  the task generating loop*/
  #pragma omp single
  {
    #pragma omp task untied
    for(i=0; i<ONEBILLION; i++)
      #pragma omp task
      process(item[i]);
  }
}
```

Summary

OpenMP core features

- Parallel construct
- Work sharing constructs
- Synchronization

WITS UNIVERSITY

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-tutorials.llnl.gov/openmp/#Introduction`
- OpenMP Application Programming Interface, `https://www.openmp.org/resources/refguides/`
- OpenMP Application Programming Interface Examples, `https://www.openmp.org/specifications/`