COSC 407
Intro to Parallel Computing

Topic 9 - Work Sharing (Sections, Scheduling and Ordered Iterations)

COSC 407: Intro to Parallel Computing

1

Outline (Asynchronous)

Previously:

- Work-sharing: parallel for construct
- Data dependency
- Single, master constructs

Today

- Sections
- Scheduling Loops (static, dynamic, guided, auto)
- Ordered Iterations
- Some examples

2

Work-Sharing Constructs

- Within a parallel region, you can decide how work is distributed among the threads.

  `for` - The for construct is probably the most important construct in OpenMP.

  `single` - Assigning a task to a single thread

  `sections` - Dividing the tasks into sections. Each section is executed by one thread.

- **Implied barrier on exit** (unless **nowait** is used). No implied barrier on entry….
- As they used in parallel region, use existing threads (do not create new threads)

Parallel Sections

- Section directive allow us to assign different sections to different threads
- **Each section is executed once**. Each thread executes zero or more sections.
  - A thread may execute zero sections if (# of sections < # of threads)
  - A thread may execute more than one section if it is *fast enough* and the implementation allows it, and/or (# of sections > # of threads)
- **No order!** It is not possible to determine which sections will be executed before which, or if two sections are executed by same thread
  - Therefore, it is important that none of the later sections depends on the results of the earlier ones
- ***There is an implicit barrier*** at the end of the **"sections"** region. ***No implicit barrier at end of each "section"***")
  - There is no implicit barrier on entry

Parallel Sections, Syntax

Within a parallel block, use the following syntax

```
#pragma omp sections [clause [clause]…]
{
        #pragma omp section
                code for section1
        #pragma omp section
                code for section2
        ...
}
```

If **not within a parallel block**, use the following syntax:

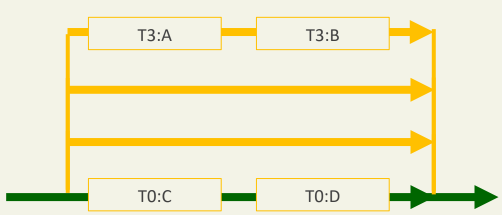
```
#pragma omp parallel sections [clause [clause]…]
  {
        #pragma omp section
        …
  }
```

The clause could be: `private`, `reduction`, `nowait`

5

Sections – Example 1

```
#pragma omp parallel
{
   #pragma omp sections
   {
        #pragma omp section
        {
          printf("T%d:A \n", omp_get_thread_num());
          printf("T%d:B \n", omp_get_thread_num());
        }
        #pragma omp section
        {
          printf("T%d:C \n", omp_get_thread_num());
          printf("T%d:D \n", omp_get_thread_num());
        }
   }
}
```



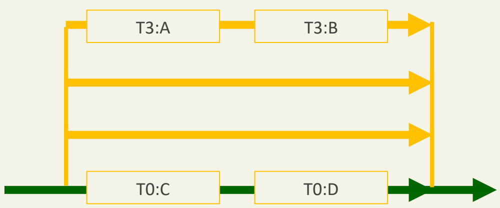
possible outcomes

T0:C	T1:A	T0:A	T0:A
T3:A	T2:C	T0:B	T0:B
T0:D	T2:D	T0:C	T1:C
T3:B	T1:B	T0:D	T1:D

6

Sections - Example -1, *cont'd*

```
#pragma omp parallel sections
{
      #pragma omp section
      {
        printf("T%d:A \n", omp_get_thread_num());
        printf("T%d:B \n", omp_get_thread_num());
      }
      #pragma omp section
      {
        printf("T%d:C \n", omp_get_thread_num());
        printf("T%d:D \n", omp_get_thread_num());
      }
}
```



possible outcomes

T0:C	T1:A	T0:A	T0:A
T3:A	T2:C	T0:B	T0:B
T0:D	T2:D	T0:C	T1:C
T3:B	T1:B	T0:D	T1:D

7

Sections - Example - 2

```
#pragma omp parallel num_threads(4)
{
   int n = omp_get_thread_num();

   printf("T%d:X \n", n);
   #pragma omp sections
   {
     #pragma omp section
     {
          printf("T%d:A \n", n);
          printf("T%d:B \n", n);
     }
     #pragma omp section
     {
          printf("T%d:C \n", n);
          printf("T%d:D \n", n);
     }
   }

   printf("T%d:Y \n", n);
}
printf("Finished");
```

Possible Output

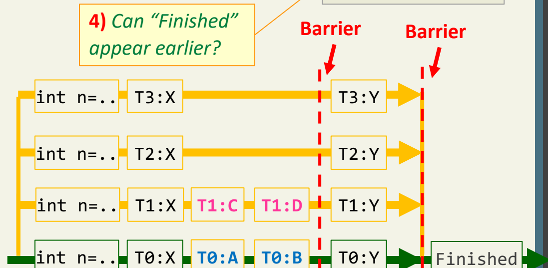
```
T1:X
T2:X
T0:X
T1:C
T0:A
T1:D
T0:B
T3:X
T1:Y
T0:Y
T2:Y
T3:Y
Finished
```

3) *Can T1:X appear here? Can T2:X appear here?*

1) *Is this an error? Shouldn't T3:X appear with the other X's above?*

2) *Why all Y's can only appear here?*

4) *Can "Finished" appear earlier?*

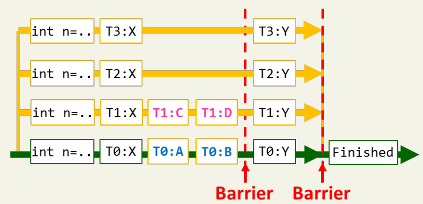


8

Sections – Example - 2

```
#pragma omp parallel
{
   int n = omp_get_thread_num();
   printf("T%d:X \n", n);
   #pragma omp sections
   {
      #pragma omp section
      {  printf("T%d:A \n", n);
         printf("T%d:B \n", n);
      }
      #pragma omp section
      {  printf("T%d:C \n", n);
         printf("T%d:D \n", n);
      }
   }
   printf("T%d:Y \n", n);
}
printf("Finished");
```

```
Possible Output
T1:X
T2:X
T0:X
T1:C
T0:A
T1:D
T0:B
T3:X
T1:Y
T0:Y
T2:Y
T3:Y
Finished
```



9

The schedule clause

Determines how iterations are distributed over threads.

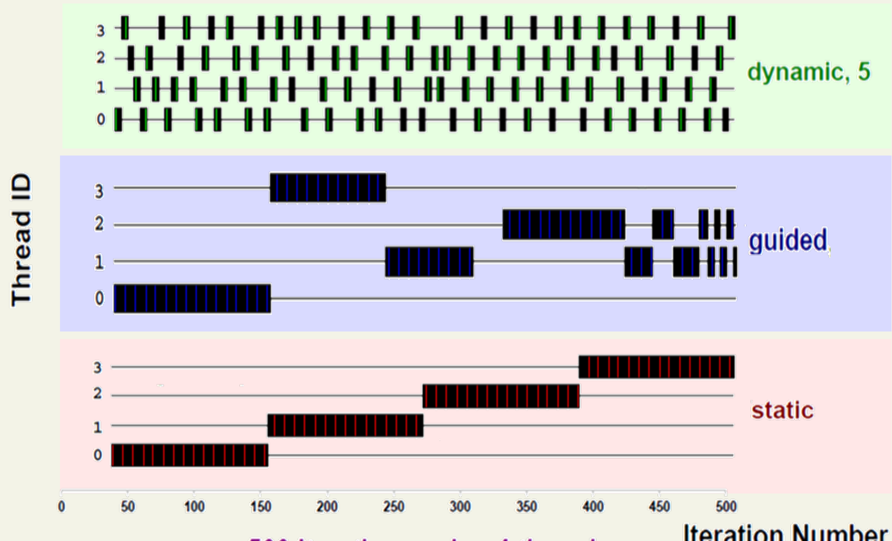
```
schedule( static|dynamic|guided|auto
[,chunksize] )
```

Aim: distribute the workload **evenly** so that processors are being used for the same amount of time.

Static is the ***default scheduling policy*** for most OpenMP implantations

12

Thread Scheduling



Ruud van der Pas, IWOMP 2010

500 iterations using 4 threads

Static Scheduling

```
schedule( static [,chunksize] )
```

- Before executing the loop, assign the iterations in blocks of size “chunk” over the threads in round-robin fashion.

Default chunk ≅ num_iterations / num_threads

Example: twelve iterations, 0, 1, . . . , 11, and 3 threads

```
schedule(static)
  T0: 0,1,2,3
  T1: 4,5,6,7
  T2: 8,9,10,11
schedule(static,4)
  T0: 0,1,2,3
  T1: 4,5,6,7
  T2: 8,9,10,11
```

```
schedule(static,2)
  T0: 0,1,6,7
  T1: 2,3,8,9
  T2: 4,5,10,11
schedule(static,1)
  T0: 0,3,6,9
  T1: 1,4,7,10
  T2: 2,5,8,11
```

Dynamic Scheduling

`schedule(dynamic [,chunksize] )`

- Iterations are broken up into chunks of chunksize.
- Each thread executes a chunk. Once done, it requests another chunk.
- This keeps going till all iterations are completed.

Default chunksize = 1

Example:

```
#pragma omp parallel for schedule(dynamic,2)
  for(int i = 0; i<8; i++)
    printf("T%d:%d\n", omp_get_thread_num(),i);
```

Possible outputs

T2:0	T1:4
T1:4	T2:0
T2:1	T0:2
T1:5	T0:3
T0:2	T1:5
T3:6	T0:6
T0:3	T2:1
T3:7	T0:7

Be careful of the ***overhead***:

once a chunk is finished, threads need to receive a new iteration counter. To overcome this problem, make sure your chunk size is reasonably large.

Guided/AUTO Scheduling

`schedule( guided | auto [,chunksize] )`

Guided

- Similar to dynamic, BUT it **starts with large chunks** then adjusts to smaller chunk sizes if the workload is imbalanced.
  - **Default**:
    - if chunksize is unspecified, chunk sizes <u>decrease down</u> to 1.
    - If chunksize is specified, it decreases down to chunksize

      But the last chuck might be smaller than chunksize.

Auto

- The compiler and/or the run-time system determine the best schedule. (OpenMP v3+)

When to Use Which?

- **Static:**
  - Use if iterations require roughly the **same amount of time**
  - It requires little overhead (mostly done at compile time)
- **Dynamic**
  - Use if iterations require **different amount of times**.
  - Allows processors the finish first to go after other chunks and hence balance the workload and keep all processors busy
  - It requires more **overhead** than static
- **Guided**
  - Use when the workload increases as we go to higher iterations.
    - i.e. when initial chunks require less time per iteration than later chunks
    - Like dynamic, there is more overhead at runtime

Experiment

We want to parallelize this loop.

```
sum = 0.0;
for(i=0; i<=n; i++)
     sum += f(i);
```

- Assume the time required by `f(i)` is proportional to value of `i`.
- Most OpenMP implementation, static scheduling is used by default.
- A better assignment might be ***dynamic*** scheduling
  - i.e. cyclic partitioning of the iterations among the threads

Experiment, *cont'd*

To get a feel for how drastically this can affect performance, assume f is:

```
double f(int i){
    int j, start = i*(i+1)/2, finish = start + i;
    double result = 0.0;

    for(j = start; j<=finish; j++)
        result += sin(j);

    return result;
}
```

- Note that time to run the inner for loop depends on `i`. for example, `f(2i)` would take twice the time of `f(i)`

Experiment*:* Results

n = 10,000

one thread (serialized)

run-time = 3.67 seconds.

two threads

Static assignment (default)

run-time = 2.76 seconds

speedup = 1.33

two threads

Dynamic assignment

run-time = 1.84 seconds

speedup = 1.99

The ordered clause

The assignment of iteration chunks to threads is unspecified and hence the **output is usually unordered**.
You can use the ordered clause to force **certain events to run in the order of iterations.**

```
#pragma omp for ordered schedule(dynamic)
for(int i=0; i<100; i++) {
     f( a[i] );
     #pragma omp ordered
     g( a[i] );
}
```

f() is done in any order and in parallel

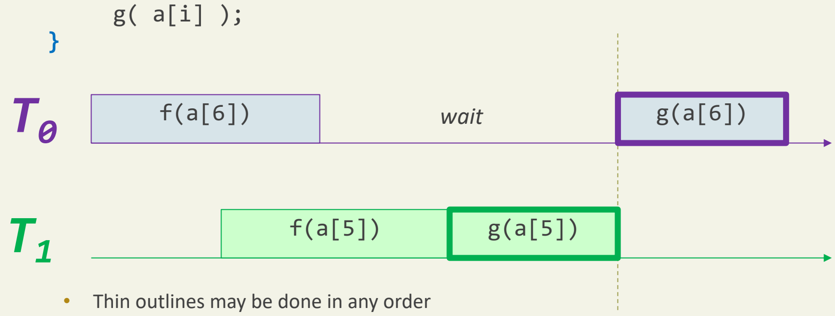
e.g. f(a[5]) and f(a[6]) may be done in parallel by two threads

g() is done strictly in order

e.g. assume a thread finishes f(a[6]) and now wants to proceed to g(a[6]). It has to check if g(a[5]) is finished. If not, the thread ***waits*** until g(a[5]) is finished.

The ordered clause

```
#pragma omp for ordered schedule(dynamic)
for(int i=0; i<100; i++) {
     f( a[i] );
     #pragma omp ordered
     g( a[i] );
}
```



- Thin outlines may be done in any order
- Thick outlines must be done in order

Rules and Cost

Rules:

- You can have **exactly one ordered block per an ordered loop**, no less and no more
- The outer for construct must contain the ordered clause

Cost

Ordering comes at an expense of wasting CPU time (waiting for other threads)

Time for (Estimating) PI…

Serial code:

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
int n = 100000000;    //100 millions
double factor = 1, sum = 0;

for(int k = 0; k < n; k++){
    sum += factor/(2*k+1);
    factor = -factor;
}

double pi = 4 * sum;
printf("%f", pi);
```

Q: Parallelize the code!

First Piece of PI

First attempt…

Will this code work??

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
int n = 100000000;    //100 millions
double factor = 1, sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int k = 0; k < n; k++){
    sum += factor/(2*k+1);
    factor = -factor;
}

double pi = 4 * sum;
printf("%f", pi);
```

Loop-carried dependency

a) Yes, it works
b) No, there is data race
c) I am super confused!

```
//possible output: 4.810167
```


25

Second Piece of PI

Second attempt…

How about this solution?

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
int n = 100000000;
double factor, sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int k = 0; k < n; k++){
    if(k%2==0)      factor = 1;
    else            factor = -1;
    sum += factor/(2*k+1);
}
double pi = 4 * sum;
printf("%f", pi);
```

a) YES, this is perfect!
b) No, there is still a problem.
c) I am still confused!

```
//possible output: 2.699575
```


26

Third Piece of PI

Third attempt… (Avoiding Data Race)

How about this one now?

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
int n = 100000000;
double factor, sum = 0;
#pragma omp parallel for reduction(+:sum) private(factor)
for(int k = 0; k < n; k++){
    if(k%2==0)      factor = 1;
    else            factor = -1;
    sum += factor/(2*k+1);
}
double pi = 4 * sum;
printf("%f", pi); //output: 3.141593
```

Ensures factor has private scope (WHY?)

Forth Piece of PI

The Easy Way…. Experiment Contender #1

Contender #1:
Use order clause to force the iterations to run in order

```
int n = 500000000;
double factor = 1, sum = 0;
double time = omp_get_wtime();
#pragma omp parallel for ordered reduction(+:sum)
for(int k = 0; k < n; k++){
    sum += factor/(2*k+1);
    #pragma omp ordered
    factor = -factor;
}
double pi = 4 * sum;
double finish = omp_get_wtime();

printf("pi = %f\ntime = %f ms", pi, 1000*(finish-time));
```

```
Output
pi = 3.141593
time = 8150 ms
```

Third Piece of PI

Experiment Contender #2

Contender #2:
Algorithm redesign (3rd Piece of PI)

```
int n = 500000000;
double factor, sum = 0;
double time = omp_get_wtime();
#pragma omp parallel for reduction(+:sum) private(factor)
for(int k = 0; k < n; k++){
    if(k%2==0)       factor = 1;
    else             factor = -1;
    sum += factor/(2*k+1);
}
double pi = 4 * sum;
double finish = omp_get_wtime();

printf("pi = %f\ntime = %f ms", pi, 1000*(finish-time));
```

```
Output
pi = 3.141593
time = 1202 ms
```


Comparison

Contender #1
Using ordered clause

```
Output
pi = 3.141593
time = 8150 ms
```

Contender #2
Redesigning the algorithm

```
Output
pi = 3.141593
time = 1202 ms
```

Conclusion/Up Next

- What we covered today (review key concepts):
  - Sections
  - Scheduling Loops (static, dynamic, guided, auto)
  - Ordered Iterations
  - Some Examples

- Next:
  - Some More Examples
    - Matrix multiplication
    - Max reduction
  - Asides and Comments on OpenMP