

Parallel and Distributed Computing

CS3006 (BDS-6A)

Lecture 11

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

07 March, 2023

Previous Lecture

- The `reduction` clause
- The `atomic` clause
- Serial and Parallel Pi using `reduction()`
- `Sections` and `section` (functional parallelism)
- Clauses: `single`, `barrier`, `nowait`, `locks`

Synchronization locks

```
/* Initialize an OpenMP lock */  
void omp_init_lock(omp_lock_t *lock);
```

```
/* Ensure that an OpenMP lock is uninitialized */  
void omp_destroy_lock(omp_lock_t *lock);
```

```
/* Set an OpenMP lock. The calling thread behaves as if it was  
suspended until the lock can be set */  
void omp_set_lock(omp_lock_t *lock);
```

```
/* Unset the OpenMP lock */  
void omp_unset_lock(omp_lock_t *lock);
```

Synchronization locks

- Locks must be:

- Initialized
- Destroyed

- Locks can be:

- Set
- Unset
- Tested

```
/** Do this only once!! */  
/* Declare lock var */  
omp_lock_t lock;  
/* Init the lock */  
omp_init_lock(&lock);
```

```
/* If another thread set the lock, I will wait */  
omp_set_lock(&lock);
```

```
/* I can do my work being sure that no-one else is here */
```

```
/* unset the lock so that other threads can go */  
omp_unset_lock(&lock);
```

```
/** Do this only once!! */  
/* Destroy lock */  
omp_destroy_lock(&lock);
```

Source:

<http://algoing.unimo.it/people/andrea/Didattica/HPC/SlidesPDF/07.%20OMP%20barriers%20and%20critical.pdf>

Synchronization locks

- `omp_init_lock()`
- `omp_set_lock()`
- `omp_unset_lock()`
- `omp_test_lock()`
- `omp_destroy_lock()`

The `private` clause

```
#pragma omp parallel for private (j)
  for (i = 0; i < M; i++)
    for (j=0; j < N; j++)
      a[i][j] = min(a[i][j], a[i][k] + tmp[j]);
```

- Direct the compiler to make one or more variables `private`.
- We need every thread to work through N values of `j` for each iteration of the `i` loop.
- If we do not make `j` `private`, all the threads try to initialize and increment the same shared variable `j` – leading to a data race.
- The private copies of the variable `j` will be accessible only inside the `for` loop. The values are undefined on loop entry and exit.

Some Useful Clauses

- A clause is an optional, additional component to a pragma
- **Private:** The private clause directs the compiler to make one or more variables private

```
int k=3;
#pragma omp parallel for default(shared) private(j) shared(k)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = MIN(a[i][j], a[i][k] + tmp);
```

Comments:

- Here the private variable *j* is undefined -
 - when this parallel construct is entered
 - when this parallel construct is exited

Some Useful Clauses

- **firstprivate:** It directs the compiler to create private variables having initial values identical to the value of the variable controlled by the master thread as the loop is entered.

```
s = complex_function();  
#pragma omp parallel for firstprivate(s) num_threads(2)  
for (i = 0 ; i < n ; i ++ ) {  
    s = s * omp_get_thread_num();  
    printf("S is %d at thread #%d\n", s, omp_get_thread_num());  
}
```


Some Useful clauses

- **lastprivate:** used to copy back to the master thread's copy of the variable, the private copy of the variable from the thread that executed the last iteration.

```
s = complex_function();  
#pragma omp parallel for private(j) firstprivate(s) lastprivate(s)  
for (i = 0 ; i < n ; i ++)  
    s +=1;  
}  
printf("s after join:%d\n", s); //value of s as it was for last iteration of the loop
```

Conditional Parallelism

- **if Clause:** The if clause gives us the ability to direct the compiler to insert code that determines at run-time whether the loop should be executed in parallel or not.
- **The clause has this syntax:** `if (<scalar expression>)`

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x) reduction(+:area) if(n>5000)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Scheduling Loops

1. **Static:** `schedule(static[, chunk-size])`

- Splits the iteration space into equal chunks of size `chunk-size` and assigns them to threads in a round-robin fashion.
- When no `chunk-size` is specified, the iteration space is split into as many chunks as there are threads (i.e., size of each is $n/total_threads$) and one chunk is assigned to each thread.
- Decision about work division is done before actually executing the code.
- Results in lower scheduling overhead. But, this can cause load-imbalance if all processors are not of same compute-capability.

Scheduling Loops

2. Dynamic: `schedule(dynamic[, chunk-size])`

- The iteration space is partitioned into chunks given by `chunk-size`
- Initially every thread is assigned a single chunk. The decision for remaining iteration chunks is done on run-time
- This means a *chunk* is assigned to threads as they become *idle*.
- This takes care of the temporal imbalances resulting from static scheduling.
- If no `chunk-size` is specified, it defaults to a single iteration per *chunk*

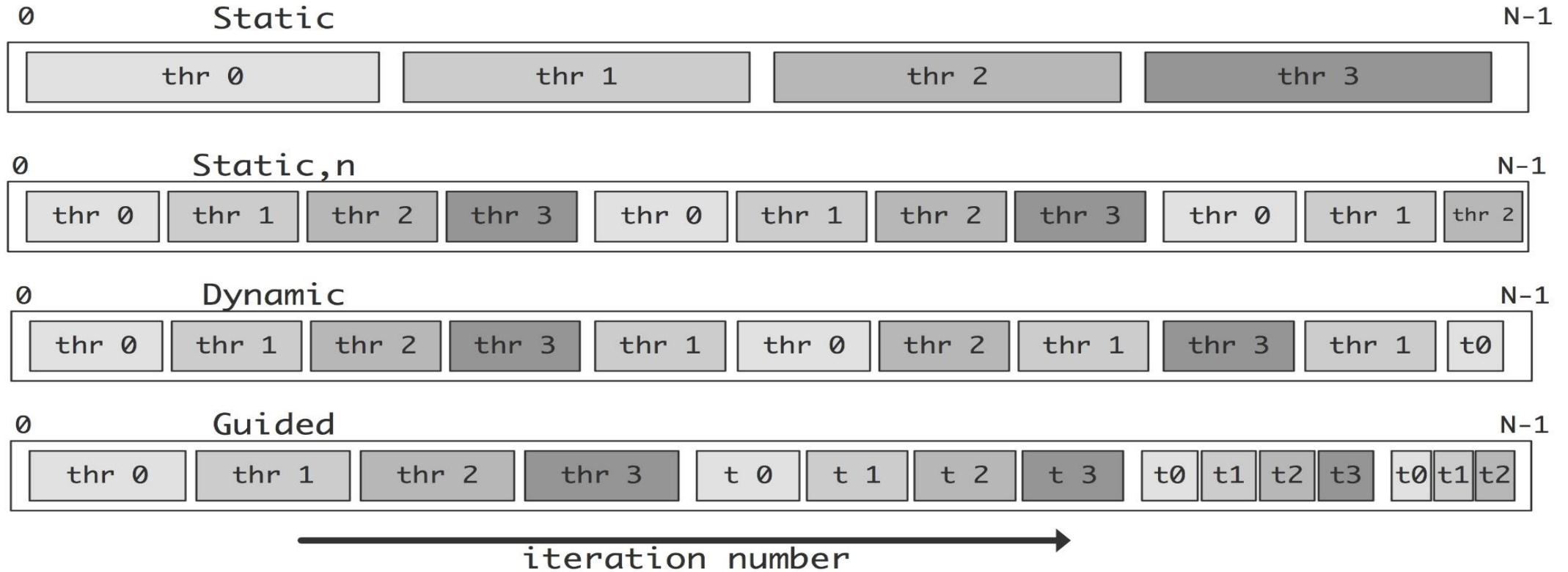
Scheduling Loops

3. Guided:

- `schedule(guided, C)`: dynamic allocation of chunks to tasks using guided self-scheduling heuristic. Initial *chunks* are bigger, later *chunks* are smaller, **minimum** chunk size is *C*.
- `schedule(guided)`: guided self-scheduling with minimum chunk size 1

4. `schedule(runtime)`: schedule chosen at run-time based on value of `OMP_SCHEDULE` environment variable.

Scheduling Loops(Summary)



Environment Variables

- OpenMP provides additional environment variables that help control execution of parallel programs
 - **OMP_NUM_THREADS**
 - **OMP_DYNAMIC**
 - **OMP_SCHEDULE**
 - **OMP_NESTED**

Environment Variables

OMP_NUM_THREADS

- Specifies the default number of threads created upon entering a parallel region.
- The number of threads can be changed during run-time using:
 - `omp_set_num_threads(int threads)` routine [OR]
 - `num_threads` clause → `num_threads(int threads)`
- Setting OMP_NUM_THREADS to 4 using bash:
`export OMP_NUM_THREADS=4`

Environment Variables

OMP_DYNAMIC

- when set to **TRUE**, allows the number of threads to be controlled at runtime. It means OpenMP will use its **dynamic adjustment algorithm** to create number of threads that may optimize system performance
 - In case of **TRUE**, the total number of threads generated may not be equal to the threads requested by using the **omp_set_num_threads()** function or the **num_threads** clause.
 - In case of **FALSE**, usually the total number of generated threads in a parallel region become as requested by the **num_threads** clause
- OpenMP routines for setting/getting dynamic status:
 - **void omp_set_dynamic(int flag); // disables if flag=0**
 - Should be called from outside of a parallel region
 - **int omp_get_dynamic(); //return value of dynamic status**

Environment Variables

OMP_DYNAMIC[dynamic.c]

The omp_get_max_threads routine returns an upper bound on the number of threads that could be used to form a new team if a parallel construct without a num_threads clause were encountered after execution returns from this routine

```
workers = omp_get_max_threads(); //can use num_procs
printf("%d maximum allowed threads\n", workers);
printf("total number of allocated cores are:%d\n", omp_get_num_procs());
omp_set_dynamic(1);
omp_set_num_threads(8);
printf("total number of requested when dynamic is true are:%d\n", 8);
#pragma omp parallel
{
    #pragma omp single nowait
    printf("total threads in parallel region1=%d:\n", omp_get_num_threads());
    #pragma omp for
    for (i = 0; i < mult; i++)
        { a = complex_func(); }
}
```

```
4 maximum allowed threads
total number of allocated cores are:4
total number of requested when dynamic is true are:8
total threads in parallel region1=4:
```

Environment Variables

OMP_DYNAMIC[dynamic.c]

```
omp_set_dynamic(0);
omp_set_num_threads(8);
printf("total number of requested when dynamic is false are:%d\n", 8);
#pragma omp parallel
{
    #pragma omp single nowait
    printf("total threads in parallel region2=%d:\n", omp_get_num_threads());
    #pragma omp for
    for (i = 0; i < mult; i++)
    {a = complex_func();}
}
```

```
total number of requested when dynamic is false are:8
total threads in parallel region2=8:
```

Environment Variables

OMP_SCHEDULE

- Controls the assignment of iteration spaces associated with *for* directives that use the runtime scheduling class
- Possible values: *static*, *dynamic*, and *guided*
 - Can also be used along with *chunk-size* [optional]
- If *chunk-size* is not specified than default *chunk-size* of 1 is used.
- Setting `OMP_SCHEDULE` to *guided* with minimum *chunk-size* of 4 using Ubuntu-based terminal:

```
export OMP_SCHEDULE= "guided, 4"
```

Environment Variables

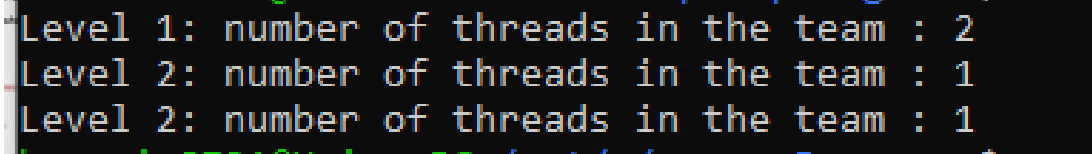
OMP_NESTED

- Default value is **FALSE**
 - While using nested parallel pragma inside another, the nested one is executed by the *original team* instead of making *new thread team*.
- When **TRUE**
 - Enables *nested parallelism*
 - While using nested parallel pragma code inside another, it makes a *new team of threads* for executing the nested one.
- Use **omp_set_nested(int val)** with non-zero value to set this variable to **TRUE**.
 - When called with '0' as argument, it sets the variable to **FALSE**

Environment Variables

OMP_NESTED [nested.c]

```
omp_set_nested(0);  
#pragma omp parallel num_threads(2)  
{  
    #pragma omp single  
    printf("Level 1: number of threads in the team : %d\n", omp_get_num_threads());  
  
    #pragma omp parallel num_threads(4)  
    {  
        #pragma omp single  
        printf("Level 2: number of threads in the team : %d\n", omp_get_num_threads());  
    }  
}
```



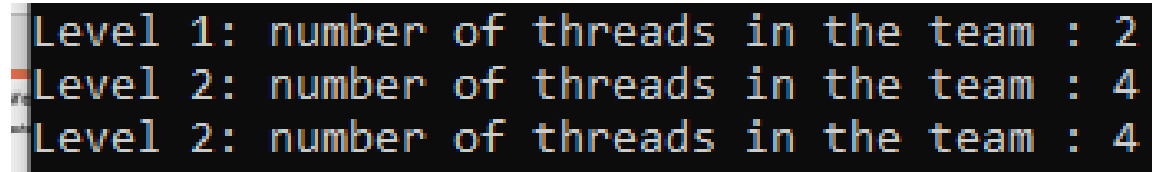
```
Level 1: number of threads in the team : 2  
Level 2: number of threads in the team : 1  
Level 2: number of threads in the team : 1
```

Environment Variables

OMP_NESTED [nested.c]

```
omp_set_nested(1);
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    printf("Level 1: number of threads in the team : %d\n", omp_get_num_threads());

    #pragma omp parallel num_threads(4)
    {
        #pragma omp single
        printf("Level 2: number of threads in the team : %d\n", omp_get_num_threads());
    }
}
```

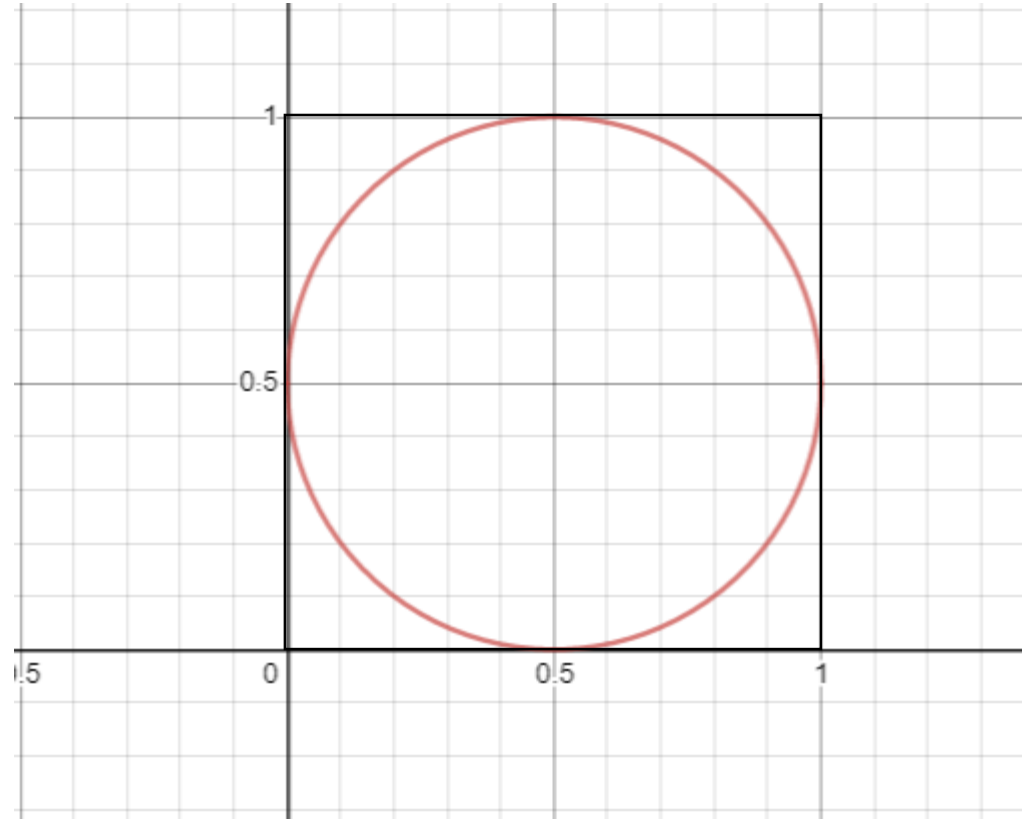


```
Level 1: number of threads in the team : 2
Level 2: number of threads in the team : 4
Level 2: number of threads in the team : 4
```

Computing Pi using Monte Carlo method

Preliminary Idea:

$$\text{Pi} = 4 \times \left(\frac{\text{points in circle}}{\text{points in square}} \right)$$



Equation for points in circle: $(x - a)^2 + (y - b)^2 < r^2$

Here $a=0.5$, $b=0.5$ and $r=0.5$

Computing Pi using Monte Carlo method

Steps

For all the random points

1. Calculate total points in the circle
2. Divide points in the circle to the points in the square
 - Total number of points are also the total number of points inside the square
3. Multiply this fraction with 4

As number of random points increases, the value of Pi approaches to real value (i.e., 3.14179.....)

Sequential Implementation

Computing Pi using
Monte Carlo method

```
int niter= 100000000;
count=0;
seed(time(0));
for (i=0; i < niter; ++i) { //10 million
    //get random points
    x = (double) random()/RAND_MAX;
    y = (double) random()/RAND_MAX;
    z = ((x-0.5)*(x-0.5))+((y-0.5)*(y-0.5));
    //check to see if point is in unit circle
    if (z<0.25) {
        ++count;
    }
}
pi = ((double) count/ (double) niter) * 4.0; //p = 4(m/n)
printf("Seq_Pi: %f\n", pi);
```

```
#pragma omp parallel shared(niter) private(i, x, y, z, chunk_size, seed) reduction(+:count) {

    num_threads = omp_get_num_threads();
    chunk_size = niter / num_threads;
    seed = omp_get_thread_num();
    #pragma omp master
    { printf("chunk_size=%ld\n", chunk_size); }
```

```
count=0;
for (i=0; i < chunk_size; i++) {
    //get random points
    x = (double) rand_r(&seed) / (double) RAND_MAX;
    y = (double) rand_r(&seed) / (double) RAND_MAX;
    z = ((x-0.5)*(x-0.5))+((y-0.5)*(y-0.5));
    //check to see if point is in unit circle
    if (z<0.25) {
        ++count;
    }
}
pi = ((double) count / (double) niter) * 4.0;
```

```
total number of allocated cores are:16
chunk_size=6250000
parallel Pi: 3.141515
Parallel time: 0.9560 seconds
Seq_Pi: 3.141745
Sequential time: 13.3521 seconds
speedup: 13.9669
```

Total points= 10 millions

```
total number of allocated cores are:16
chunk_size=62500000
parallel Pi: 3.141598
Parallel time: 8.5668 seconds
Seq_Pi: 3.141576
Sequential time: 132.0383 seconds
speedup: 15.4128
```

Total points= 100 millions

Computing Pi using the Monte Carlo method

(Parallel construct [parallel_pi.c])

More Detailed Discussion

- Full Example Online:
http://www.umsl.edu/~siegelj/cs4790/openmp/pimonti_omp.c.HTML
- Further Reading (optional):
 - <https://1drv.ms/p/s!Apc0G8okxWJ12jlUANAQsYO-JVdx?e=VixgYX> (just slide 1-9)
 - https://passlab.github.io/CSCE569/notes/lecture04-07_OpenMP.pdf
 - <https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf>