Parallel and Distributed Computing

CS3006

Lecture 10

OpenMP-III

6th April 2022

Dr. Rana Asif Rehman

Review of OpenMP Clause List

- Private
  - firstprivate, lastprivate
- Shared
- Default
  - private, shared, none
- Reduction
- If clause
- Schedule
  - Static, dynamic, guided, runtime
- nowait

Synchronization in OpenMP

Barrier Directive

- On encountering this directive, all threads in a team wait until others have caught up, and then release

```
#pragma omp barrier
```

Single Directive

- A single directive specifies a structured block that is executed by a single (arbitrary) thread in parallel region
- Implicit barrier

#pragma omp single [clause list]
structured block

Master Directive

- The master directive is a specialization of the single directive in which only the master thread executes the structured block
- No implicit barrier

```
#pragma omp master
structured block
```

Critical Sections
(#pragma omp critical)

- A Critical Section is a code segment that has a shared variable and need to be executed as an atomic action.
  - It means that in a group of cooperating processes/threads, at a given point of time, **only one process must be executing its critical section**
- Forces threads to be mutex (mutually exclusive)
  Only one thread at a time executes the given code section

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
   x += (i+0.5)/n; //can be calculated independently
   area += 4.0/(1.0 + x*x); //requires mutex lock.
}
pi = area / n;
```

Critical Sections
(#pragma omp critical)

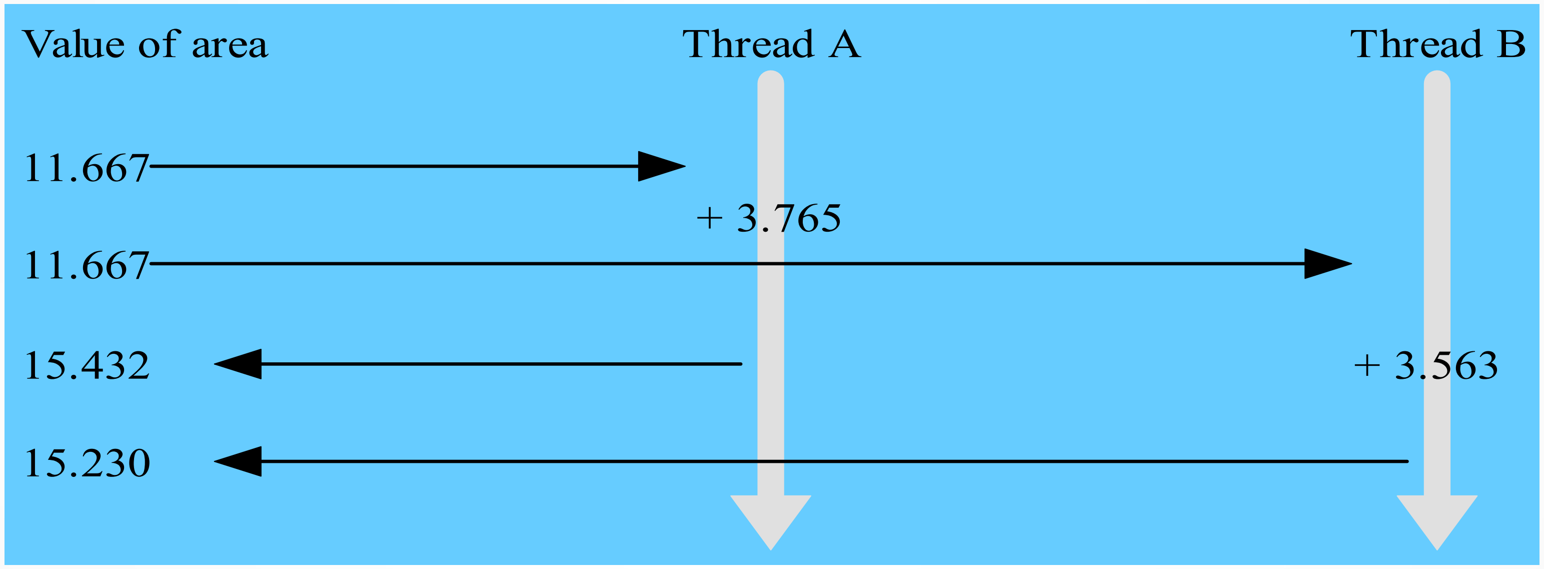
- If we simply parallelize the loop... A **race condition** may occur

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
    x = (i+0.5)/n;
    area += 4.0/(1.0 + x*x); //not atomic
}
pi = area / n;
```

Critical Sections

(#pragma omp critical)

Race Condition



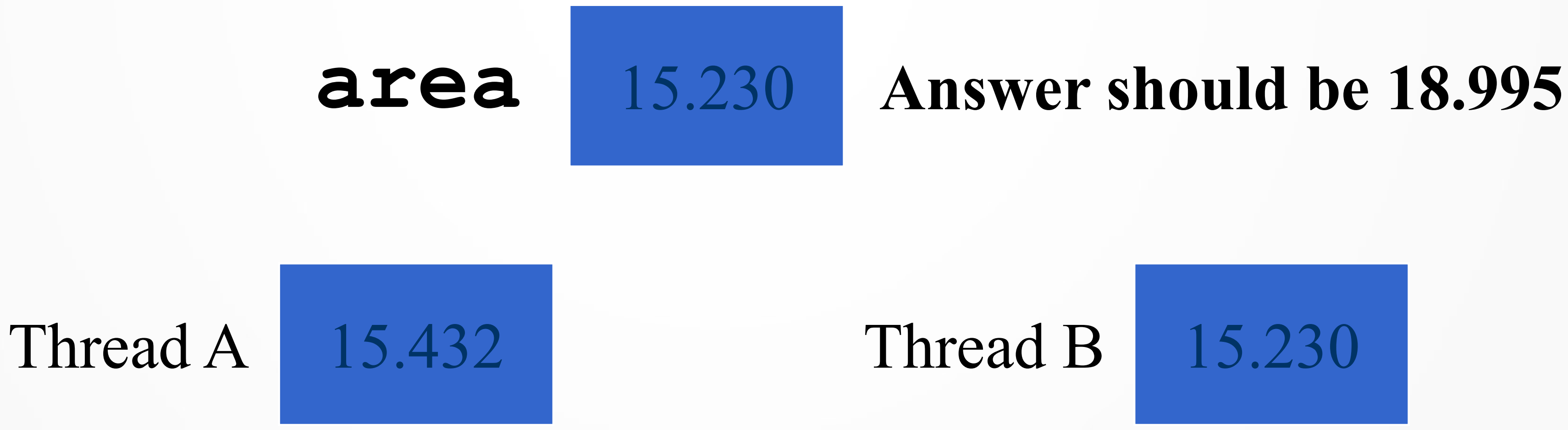
- Thread A reads value of *area* first
- Thread B reads value of *area* before A can update its value
- Thread A updates value of *area*
- Thread B ignores update by A and writes its incorrect value to *area*

Critical Sections

(#pragma omp critical)

Race Condition

- A race condition is created when one process may "race ahead" of another and overwrite the change made by the first process to the shared variable



```
area += 4.0/(1.0 + x*x)
```

Critical Sections
(#pragma omp critical)

- **Critical section**: a portion of code that only thread at a time may execute
    - We denote a critical section by putting the pragma

      `#pragma omp critical [(name)]`
- Optional identifier *name* can be used to identify a critical region
- Solves the problem but, as only one thread at a time may execute the statement; it becomes sequential code

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

Atomic Directive

- The atomic directive specifies that the single memory location update should be performed as an atomic operation

#pragma omp atomic

Update instruction e.g., x++

Environment Variables in OpenMP

Environment Variables in OpenMP

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

Environment Variables in OpenMP

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 in OpenMP

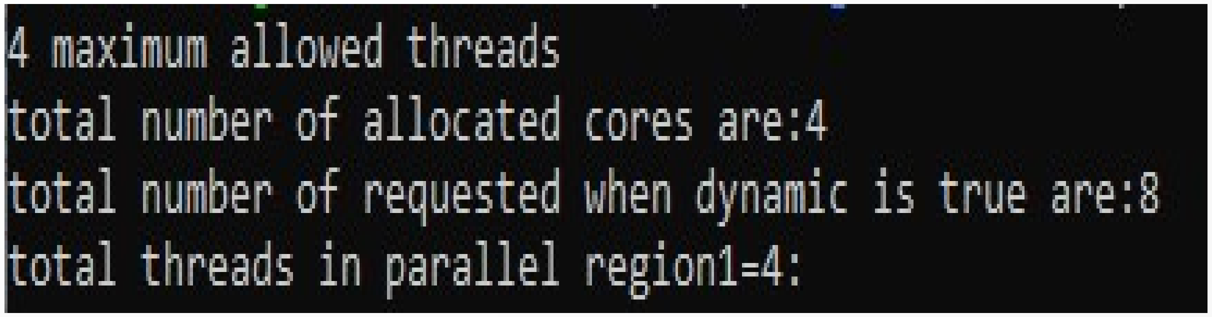
OMP_DYNAMIC

- 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
  - Incase of TRUE , 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.
  - Incase of FALSE, usually total no. 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 in OpenMP

OMP_DYNAMIC[dynamic.c]

```
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();}
}
```

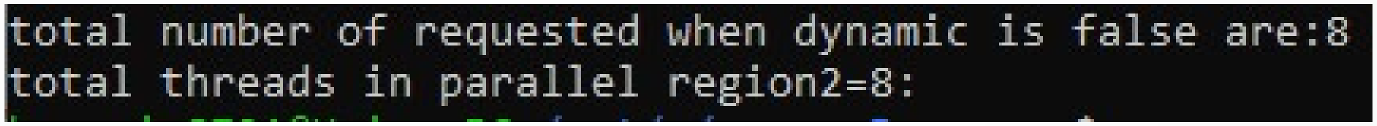


```
4 maximum allowed threads
total number of allocated cores are:4
total number of requested when dynamic is true are:8
total threads in parallel region1=4:
```

Environment Variables in OpenMP

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 in OpenMP

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 in OpenMP

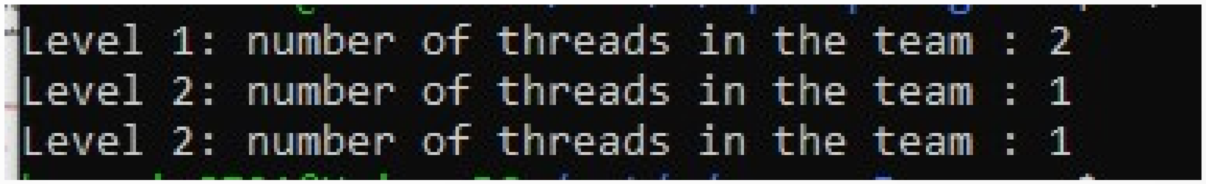
OMP_NESTED

- 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 set the variable to FALSE

Environment Variables in OpenMP

OMP_NESTED[nested.c]

```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());
    }
}
```

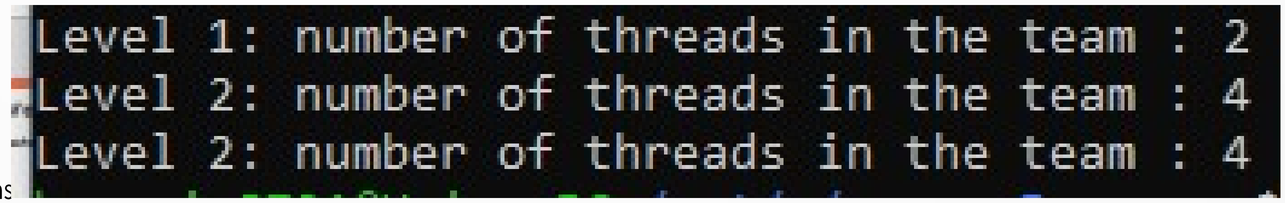


```
Level 1: number of threads in the team : 2
Level 2: number of threads in the team : 1
Level 2: number of threads in the team : 1
```

Environment Variables in OpenMP

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
```

Example

Computing Pi using Monti Carlo method

Preliminary Idea:

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

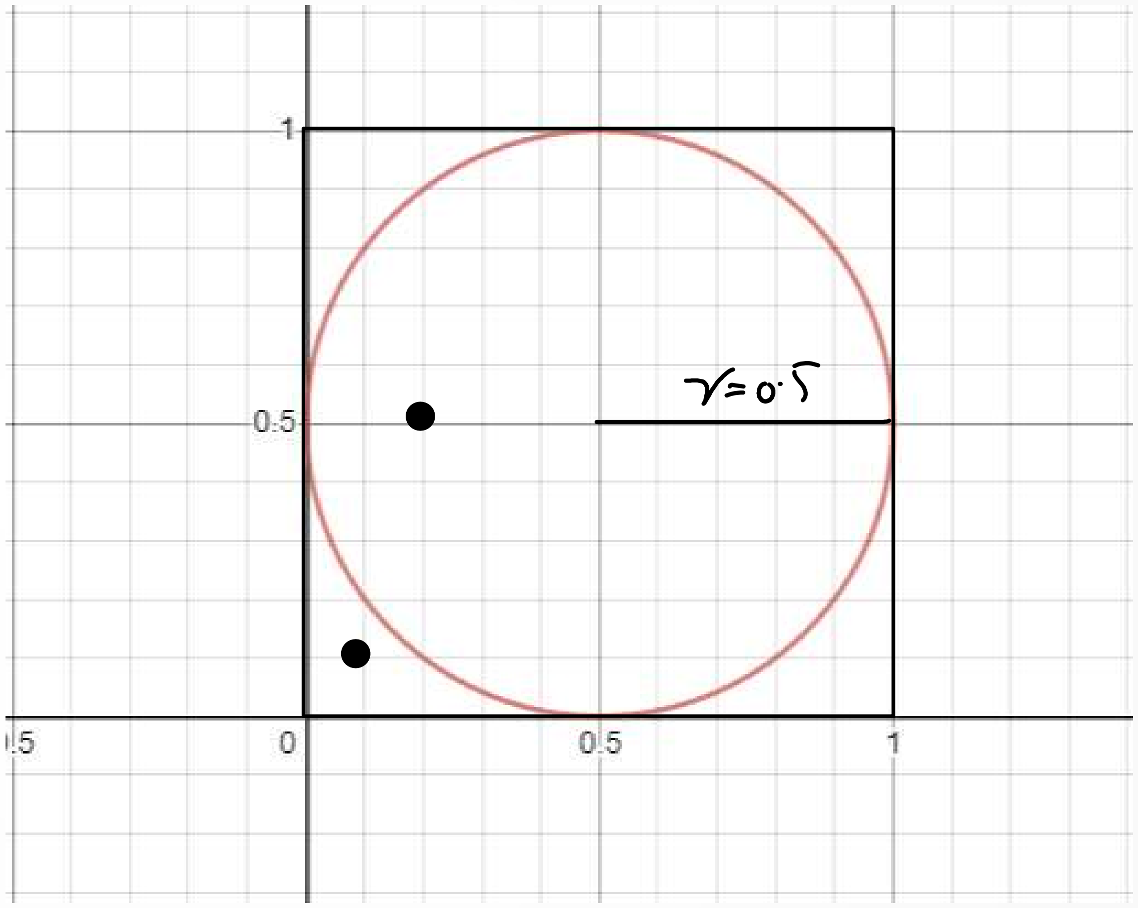
<u>prof</u>

$$A_c = \pi r^2$$

$$A_s = (2r)^2 = 4r^2$$

$$f = \frac{A_c}{A_s} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

$$\Rightarrow \pi = 4 \times f$$



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

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

Computing Pi using Monti 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…..)

Computing Pi using Monti Carlo method

Sequential Implementation

```
     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);
```

Computing Pi using Monti Carlo method

(Parallel construct [parallel_pi.c])

```
#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;
```

Parallelizing linked lists

Consider the following code:

```
current=head;
while(current->next != NULL){
    complex_func(current->key);  //complex consumer func
    current=current->next;
}
```

- Assume that complex function can be computed foreach key value independently
- The code can't be parallelized directly as:
  - We don't have omp constructs to parallelize while loops and equivalent *for* loop don't have canonical form.
  - This is because we don't know number of iterations in advance
  - If we simply put '*omp parallel* pragma' before while, program semantics will not be assured

Parallelizing linked lists [Naïve idea:1 with logical error]

Consider the following code:

```
      current=head;
#pragma omp parallel firstprivate(current)
{
     while(current-> next != NULL){
          complex_func(current->key);  //complex consumer func
          current=current-> next;
      }
}
```

- Creates team of threads, each with private ‘current’ variable.
- Each thread will execute for all the nodes in the list
- This means every thread will perform work equal to sequential code
- No speedup achieved, this will rather increase execution time

Parallelizing linked lists [Naïve idea:2 with logical error]

Consider the following code:

```
      current=head;
#pragma omp parallel shared(current)
{
     while(current-> next!=NULL){ //line 1
          complex_func(current->key);  //complex consumer func
          current=current-> next;   //line 3
      }
}
```

- Creates team of threads sharing same ‘current’ variable.
- For first while iteration, complex_func may be called by **each thread with same key-value**.
- Semantics/atomics will not be ensured (i.e., multiple threads executing line-3 can change line-1 result for other threads)
- So, output may not be as assumed

Parallelizing linked lists [Naïve but Correct parallelization]

Observations:

1. We don’t know in advance the number nodes in the list
2. We also don’t know how to access all the nodes parallelly from the list. This because the linked-list can only be accessed sequentially

3. We can parallelize it using the following steps
   1. Count number of nodes in the list →call it ‘C’
   2. Allocate a dynamic array of pointers-to-list of size ‘C’. Now using loop, copy address of ith node to the ith element in the pointers-array.
   3. Now we can use for-loop that can iterate on this array of pointers. Furthermore, this for-loop can also be parallelized

Parallelizing linked lists [Naïve but Correct parallelization]

1. **Count number of nodes in the list →call it 'C'**

//struct LIST{ int key; LIST* ptr; } list;

```
int C=0;  LIST *p =head;
//Here assume head is pointer to the start of the list.
while(p != NULL){
   p=p->next;
   C++;
}
```

Parallelizing linked lists [Naïve but Correct parallelization]

2. Allocate a dynamic array of pointers-to-list of size ‘C’ and using loop, copy address of ith node to the ith element in the pointers-array

```
LIST **Parray = new LIST* [C];


p =head;
while(p != NULL){
    Parray[i]=p;
    p=p->next;
}
```

Parallelizing linked lists [Naïve but Correct parallelization]

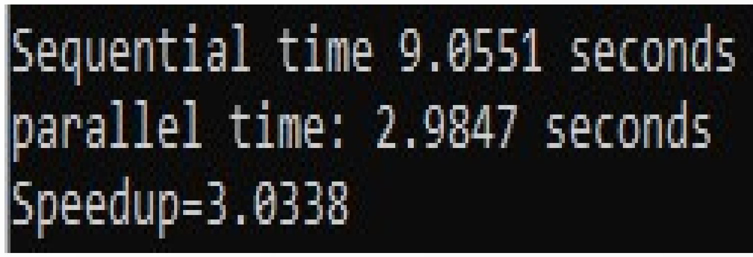
3. Now we can use for-loop that can iterate on this array of pointers. Furthermore, this for-loop can also be parallelized

```
#pragma omp parallel for schedule(static,1)
for(i=0;i<C;i++){
    complex_func(Parray[i]->key);
}
```

- This method can result in speedups only if tasks are complex enough to overcome the data-movement costs.
- Usually, data-movements are more costly than the computations
  - **So, we need to devise another solution**

Parallelizing linked lists [A relatively better implementation]

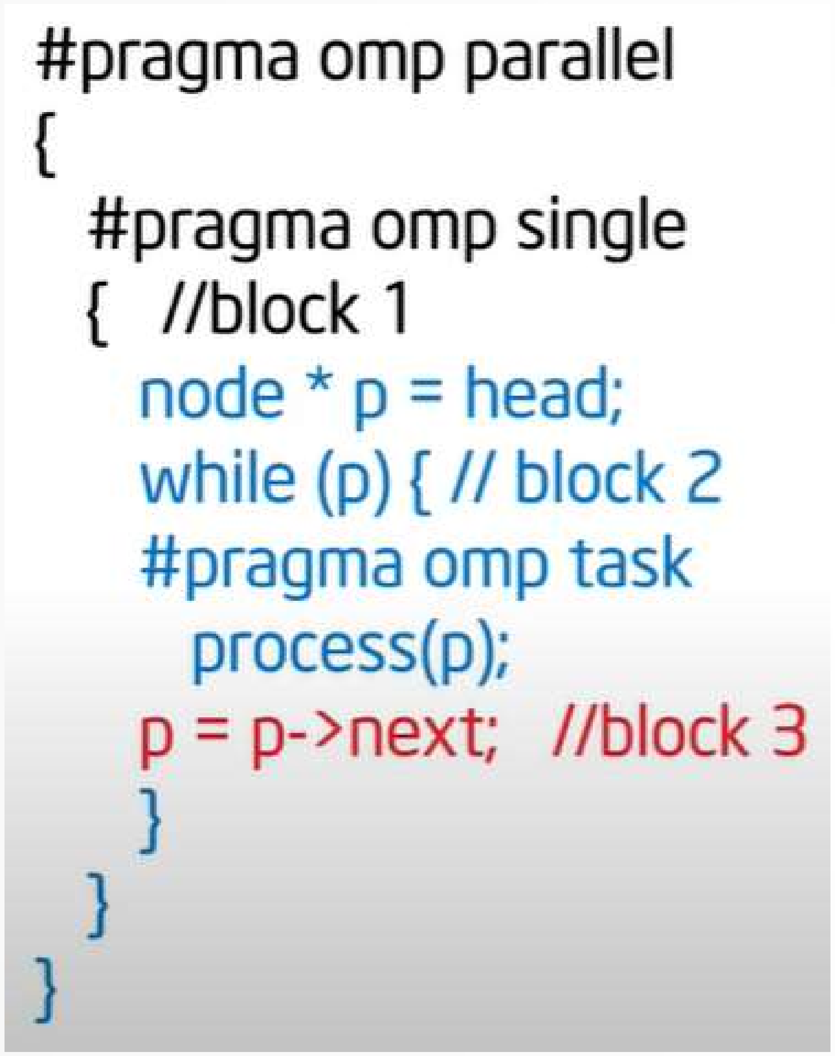
```
//omptask.c and tasktime.c //execute using g++
#pragma omp parallel
 {
    #pragma omp single //single process will go into the region
    {
        current=head;
         while(current->ptr!=NULL){
//following line creates a task and adds to logical task pool.
              #pragma omp task firstprivate(current)
              complex_func(current->key);

              current = current->ptr;
            }
    }
 }
```



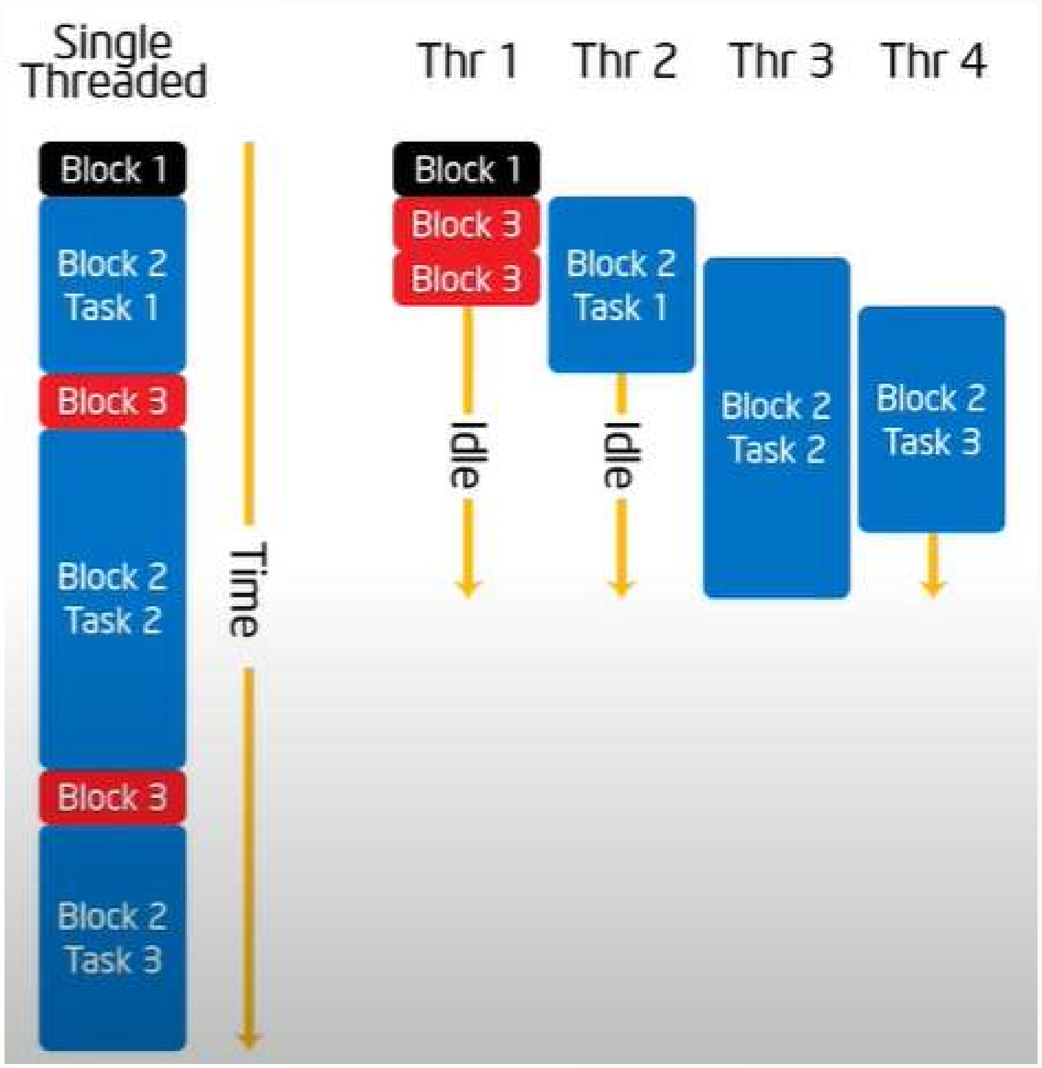
```
Sequential time 9.0551 seconds
parallel time: 2.9847 seconds
Speedup=3.0338
```

Total threads=4
Total complex itters= 100 Million
List size= 10 nodes

Parallelizing linked lists [omp task illustration]



```
#pragma omp parallel
{
  #pragma omp single
  {   //block 1
    node * p = head;
    while (p) { // block 2
    #pragma omp task
      process(p);
    p = p->next;   //block 3
    }
  }
}
```



Questions

References

1. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (2017). *Introduction to parallel computing*. Redwood City, CA: Benjamin/Cummings.