Parallel and Distributed Computing

CS3006

Lecture 9

OpenMP-II

6th March 2024

Dr. Rana Asif Rehman

Review of OpenMP Library Functions

- **Controlling Number of Threads and Processors**
  - void omp_set_num_threads (int num_threads);
  - int omp_get_num_threads ( );
  - int omp_get_max_threads ( );
  - int omp_get_thread_num ( );
  - int omp_get_num_procs ( );
  - int omp_in_parallel ( );
- **Controlling and Monitoring Thread Creation**
  - void omp_set_dynamic (int dynamic_threads);
  - int omp_get_dynamic ( );
  - void omp_set_nested (int nested);
  - int omp_get_nested ();

OpenMP

#pragma omp directive [clause list]

OpenMP Directives

- #pragma omp parallel
- #pragma omp for

One more thing to note

Difference between *omp for* **and** *omp parallel*

```
#pragma omp parallel
{
  #pragma omp for
  for (i = 0 < i < n; i++) {
   //omp for schedules/distributes itterations between the threads
        /* body of parallel for loop */
   }
}
```

is same as

```
#pragma omp parallel for
   for (i = 0 < i < n; i++) {
        /* body of parallel for loop */
    }
```

Some Useful Clauses in OpenMP

- 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 in OpenMP

- **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:**

consider the following code

```
s = complex_function();
#pragma omp parallel for private(j) firstprivate(s)
for (i = 0 ; i < n ; i ++) (
    s  += 1
}
printf("s after join:%d\n",s); //undefined value
```

Some Useful Clauses in OpenMP

- **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
```

Reduction clause

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to **`parallel for`** pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop
- The reduction clause has this syntax:
  `reduction (<op> :<variable>)`

- **Operators**
  - + Sum
  - * Product
  - & Bitwise and
  - | Bitwise or
  - ^ Bitwise exclusive or
  - && Logical and
  - || Logical or

Reduction clause

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

Conditional Parallelism Clause

- **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 (a clause)

- **Scheduling** the loops means dividing number of iterations between the processes.
- Syntax of schedule clause

```
schedule (<type>[,<chunk> ])
```

- Schedule type is required but, chunk size optional

- A chunk is a contiguous range of iterations
  - Increasing chunk size reduces scheduling overhead and may increase cache hit rate [due to operations on contiguous memory locations]
  - Decreasing chunk size allows finer balancing of workloads

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/tot.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, can cause load-imbalance if all processors are not of same compute-capability.

Scheduling Loops

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

Example when reducing chunk size improves load-balancing

```
#pragma omp parallel for private (j) schedule(static, 1)
   for (i = 0; i < n; i++)
      for ( j = i; j < n; j++)
         a[i][j] = complex_func(i,j);
```

Scheduling Loops

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

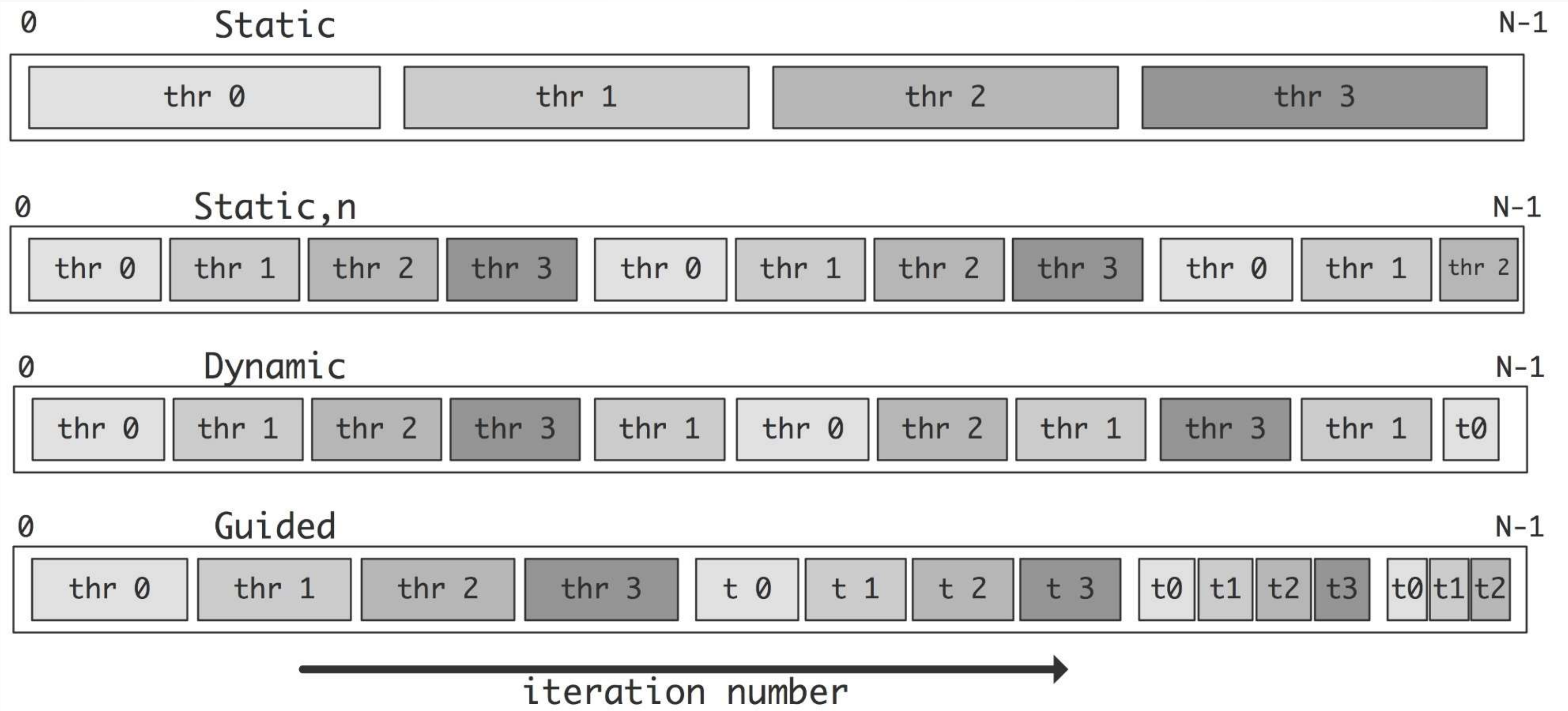
- The iteration space is partitioned into chunks given by chunk-size
- Initially every thread is assigned single chunk. The decision for remaining iteration chunks is done on run-time
- This means chunk is assigned to threads as they become idle.
- This takes care of the temporal imbalances resulting 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 env variable.

Scheduling Loops(Summary)



No Wait Clause

- In order to avoid implicit barrier
- A thread can easily move to next after completed its assign task/iterations

#pragma omp parallel for nowait

Functional / Task Parallelism in OpenMP

Functional/Task Parallelism

#pragma omp sections [clause list]

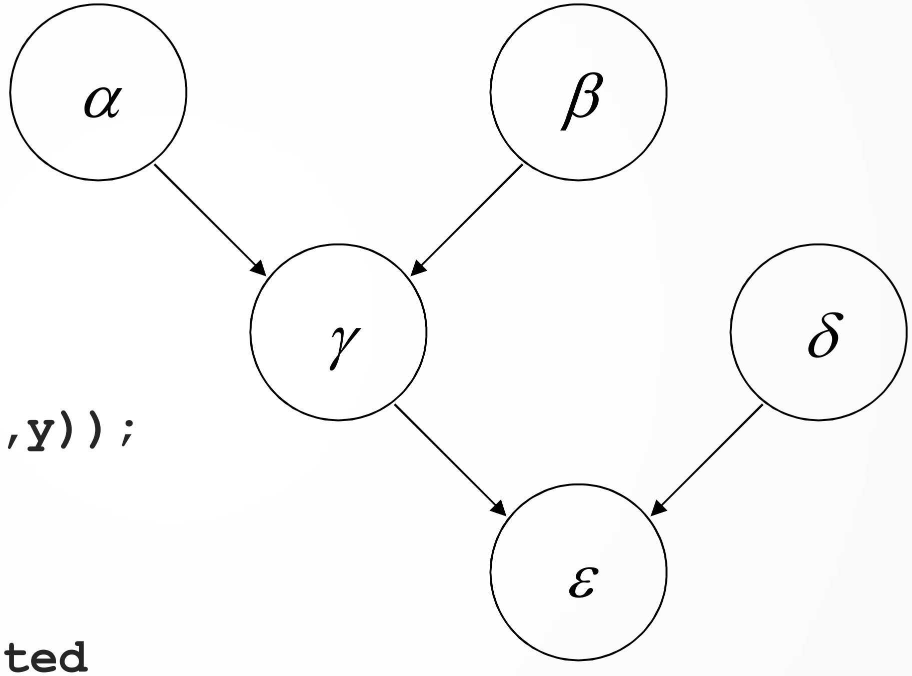
Functional/Task Parallelism

- If your code is based on different segments or sections that can be executed in parallel.
- Also known to as task parallelism

```
  v = alpha();
  W = beta();
  x = gamma(v,w);
  y = delta();

printf("%6.2f\n",epsilon(x,y));
```

- **Can execute alpha, beta, delta parallelly**
- **Remaining ones are executed sequentially according to the dependency.**



parallel sections, section pragmas

```
#pragma omp parallel sections
   {
      #pragma omp section //[optional for 1st block]
      v = alpha();
      #pragma omp section
      W = beta();
      #pragma omp section
      y = delta();
   }
 x = gamma(v,w);
 printf("%6.2f\n", epsilon(x,y));
```

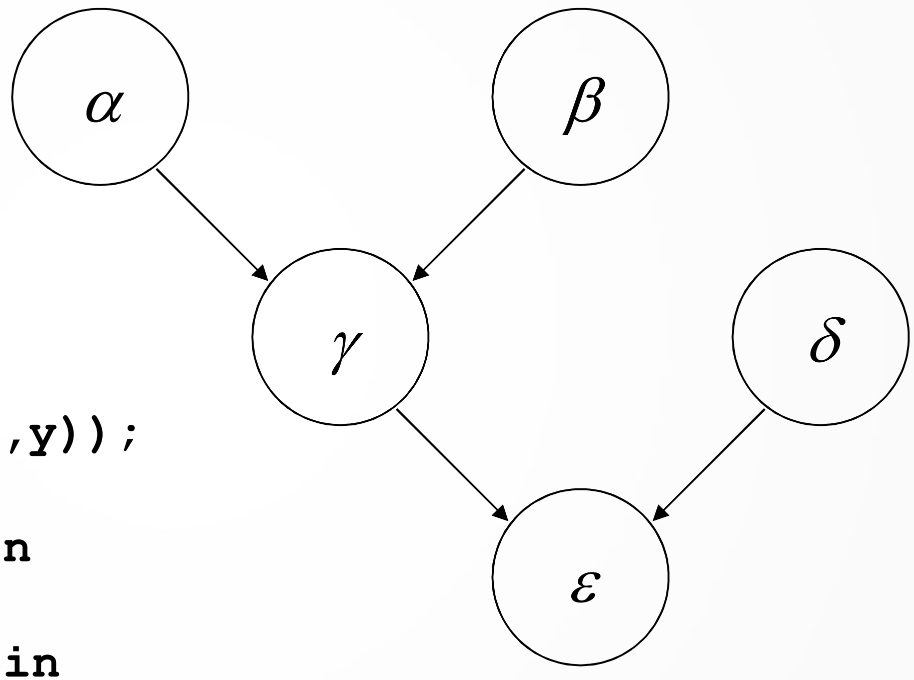
- #pragma omp parallel sections creates a team of threads which executes the sections in the region parallelly
  - Sections that can be executed parallel are preceded by 'omp section' pragma.

Functional Parallelism

Another approach

```
v = alpha();
W = beta();
x = gamma(v,w);
y = delta();

printf("%6.2f\n",epsilon(x,y));
```



- **Execute alpha and beta in parallel.**
- **Execute gamma and delta in parallel**

omp sections **pragma**

- Appears inside a parallel block of code
- This pragma distributes enclosed sections among the threads in the team

- The difference between *omp parallel sections* and *omp sections* is that,
  - ***Omp parallel sections*** generate its own team of threads
  - While simple ***omp sections*** pragma uses existing team of threads and distributes section among the threads

- If multiple sections pragmas are inside one parallel block, may reduce fork/join costs

sections pragma

```
#pragma omp parallel num_threads(2)
   {
      #pragma omp sections
      {
        #pragma omp section //optional
             v = alpha();
        #pragma omp section
           w = beta();
      } // here an implicit barrier exists

      #pragma omp sections
      {
           x = gamma(v, w);
        #pragma omp section
           y = delta();
      }
   }
   printf ("%6.2f\n", epsilon(x,y));
```

Questions

References

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