OPENMP WORKSHOP

Argonne NATIONAL LABORATORY

QUICK OVERVIEW OF OPENMP

JOSE MONSALVE

April 26th 2019 – Chicago IL

OVERVIEW

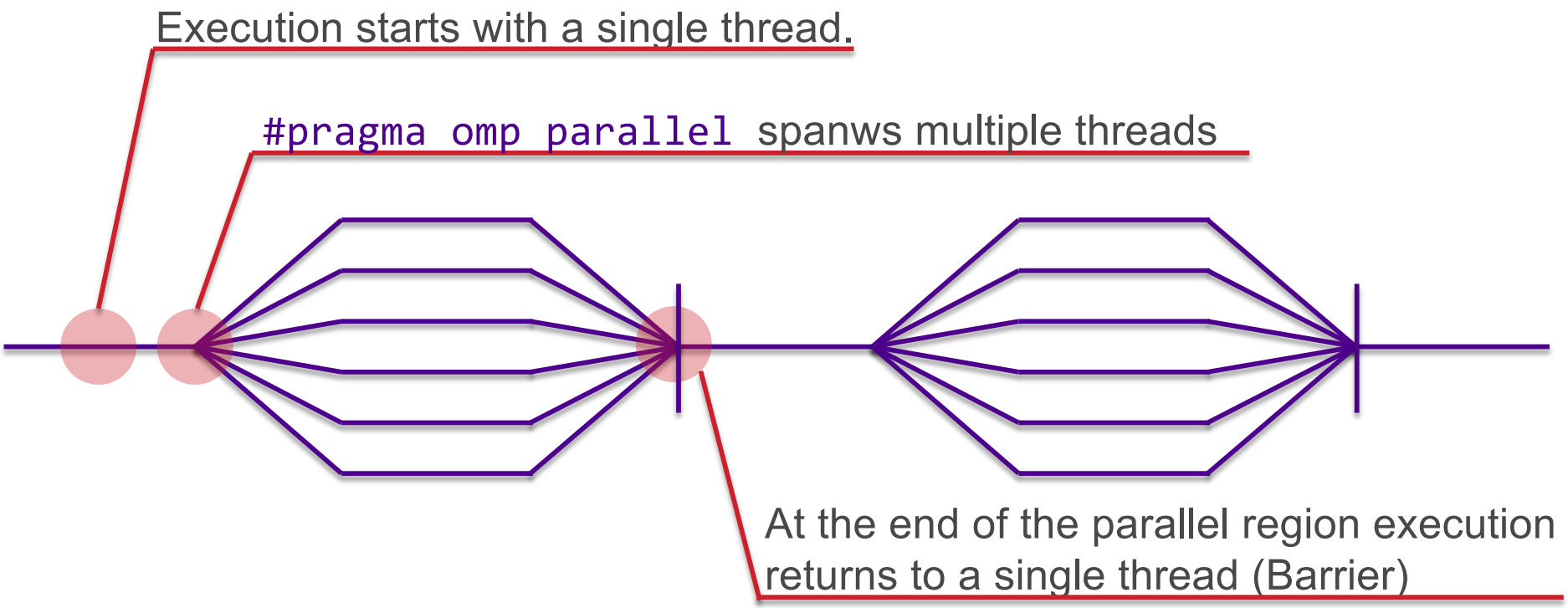
An introduction to OpenMP

1. OpenMP Programming model
   - Directives and clauses
2. OpenMP Memory Model
   - Directives and clauses
3. Tasking Model

THE OPENMP PROGRAMMING MODEL

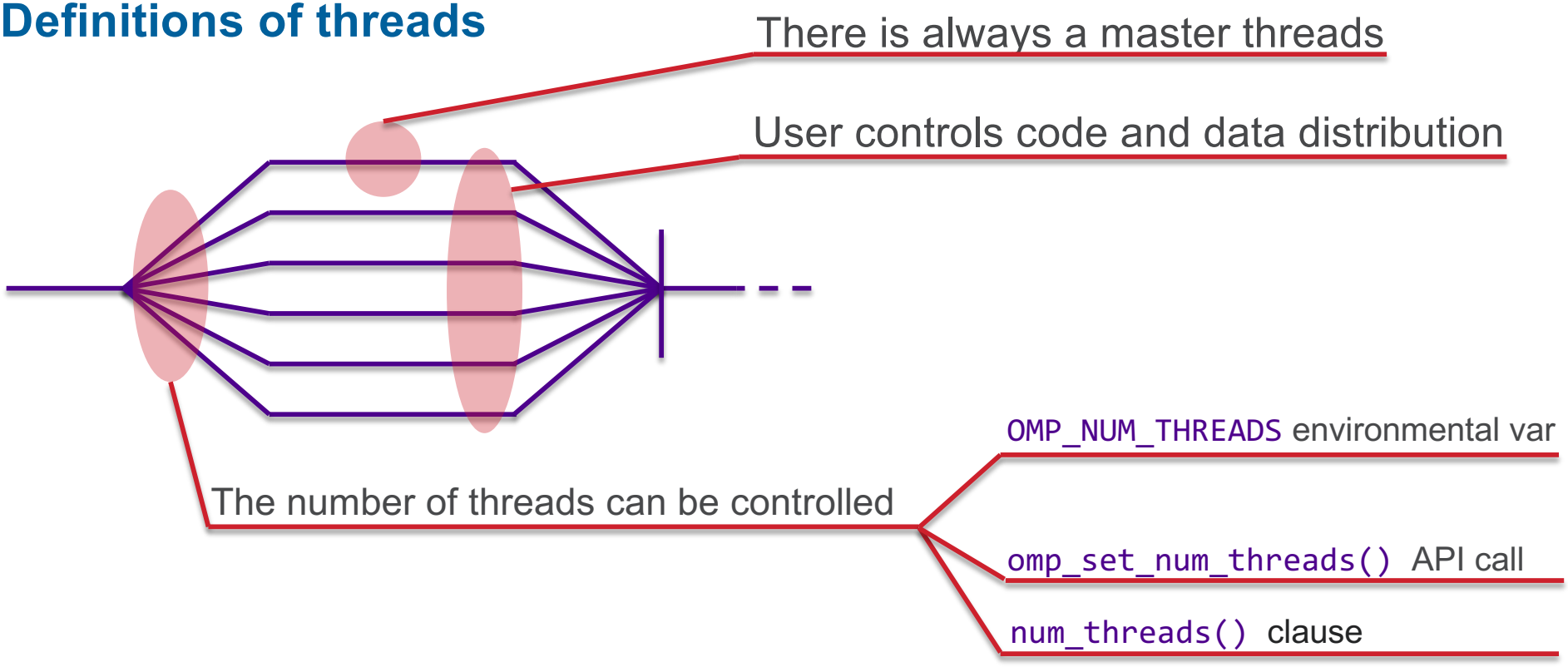
FORK AND JOIN MODEL

Parallel regions



FORK AND JOIN MODEL

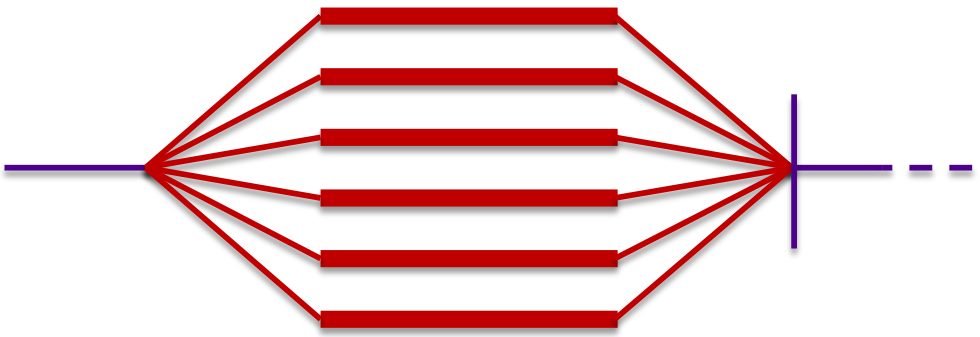
Definitions of threads



REVIEW OF OPENMP DIRECTIVES

FORK AND JOIN MODEL

Parallel directive



- Same code is executed by all the threads
- Each thread has its own identifier
- There is private and shared memory
- Unless *nowait* clause is used, there is a barrier at the end of the parallel region

file: parallel.c

```
1 #pragma omp parallel
2 {
3     printf("Hello, I am %d\n",omp_get_thread_num());
4 }
```

FORK AND JOIN MODEL

Master directive

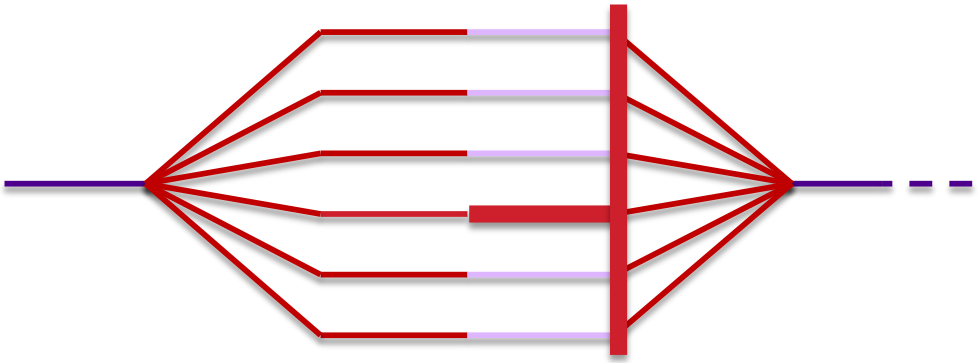
- All the threads execute the parallel region
- But only the master threads execute line 6
- There is no barrier at the end of the master region

file: master.c

```
1 #pragma omp parallel
2 {
3     printf("Hello, I am %d\n",omp_get_thread_num());
4 #pragma omp master
5     {
6         printf("This only once from %d\n",omp_get_thread_num());
7     }
8 }
```

FORK AND JOIN MODEL

Single directive



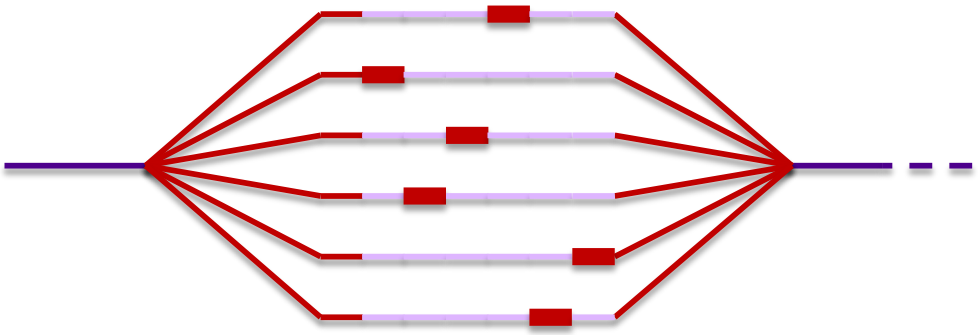
- All the threads execute the parallel region
- But only the a single threads execute line 6
- It can be a thread different than the master
- Unless *nowait* clause, there is a barrier after the single region

file: single.c

```
1 #pragma omp parallel
2 {
3     printf("Hello, I am %d\n",omp_get_thread_num());
4 #pragma omp single
5     {
6         printf("This only once from %d\n",omp_get_thread_num());
7     }
8 }
```

FORK AND JOIN MODEL

Critical directive



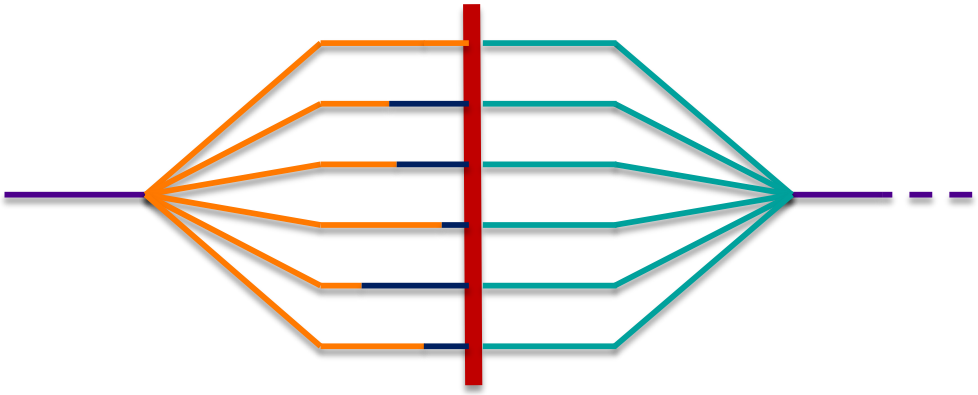
```
1 #pragma omp parallel
2 {
3     printf("Hello, I am %d\n", omp_get_thread_num());
4 #pragma omp critical
5     {
6         someCriticalWork();
7     }
8 }
```

file: critical.c

- All the threads access the critical region at some point
- But only a single threads at a time executes the thread unsafe work at line 5
- Guarantees mutual exclusion

FORK AND JOIN MODEL

Barrier directive



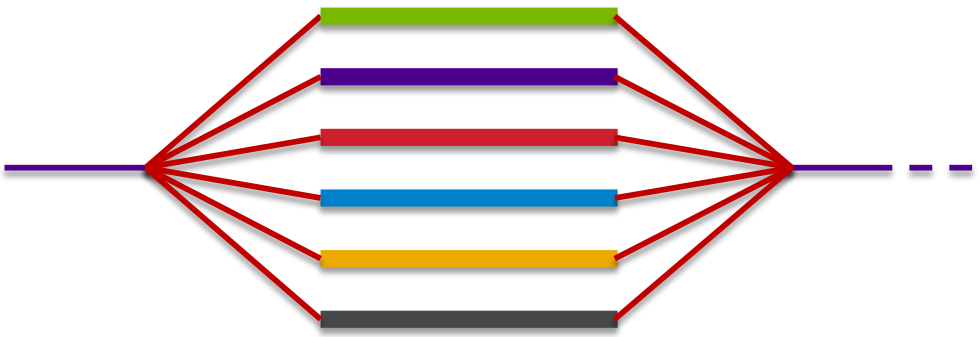
- Global synchronization of threads
- All the threads executed all the work before the barrier, and wait for everyone to reach it.
- All “hi from” messages should be printed before all “bye from” messages

file: barrier.c

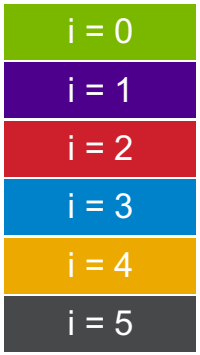
```
1 #pragma omp parallel
2 {
3     printf("Hi from %d\n", omp_get_thread_num());
4 #pragma omp barrier
5     printf("Bye from %d\n", omp_get_thread_num());
6 }
```

FORK AND JOIN MODEL

Parallel for/do loop directive



Iteration space:



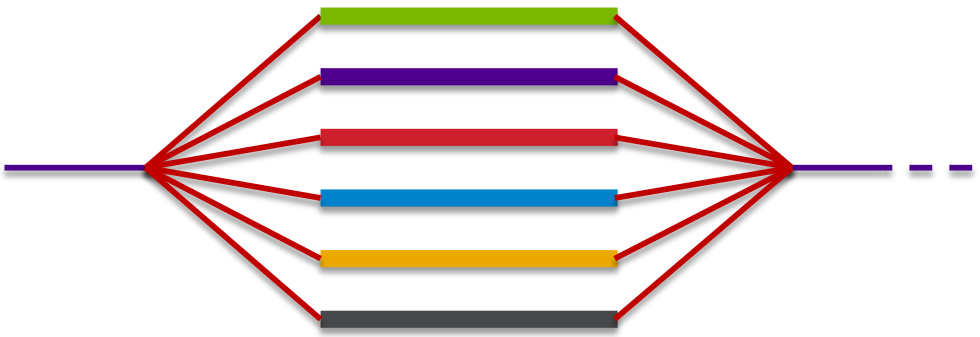
- Loop is executed in parallel
- Each thread gets a chunk of the iteration space
- **How to distribute the iterations?**

file: parallel_for.c

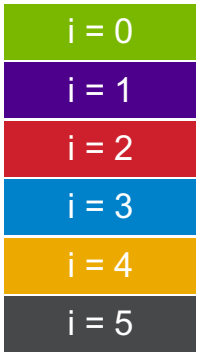
```
1 #pragma omp parallel for
2 for (int i = 0; i < 6; i++)
3 {
4     printf("Hi from %d iteration %d \n", omp_get_thread_num(), i);
5 }
```

FORK AND JOIN MODEL

Parallel for/do loop directive



Iteration space:



- Loop is executed in parallel
- Each thread gets a chunk of the iteration space
- **How to distribute the iterations?**
  - A: *schedule()* clause

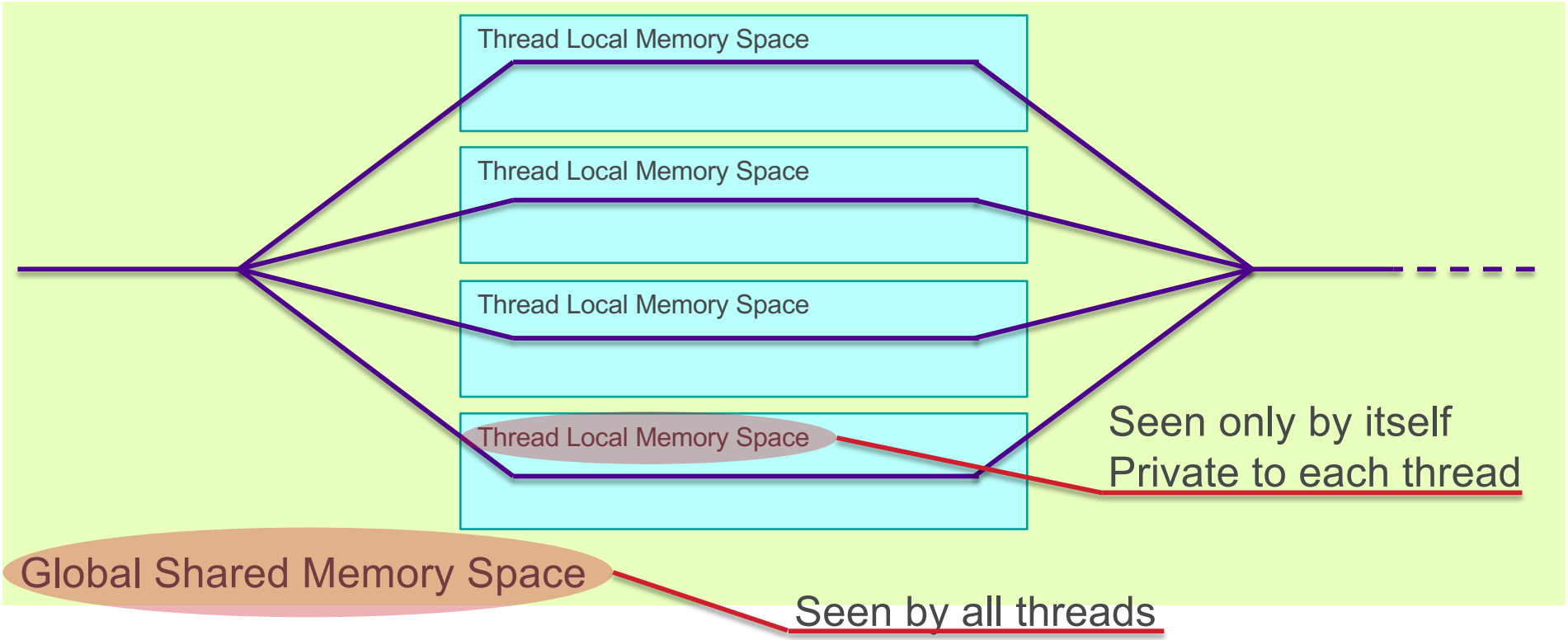
file: parallel_for.c

```
1 #pragma omp parallel for
2 for (int i = 0; i < 6; i++)
3 {
4     printf("Hi from %d iteration %d \n", omp_get_thread_num(), i);
5 }
```

OPENMP MEMORY MODEL AND CLAUSES

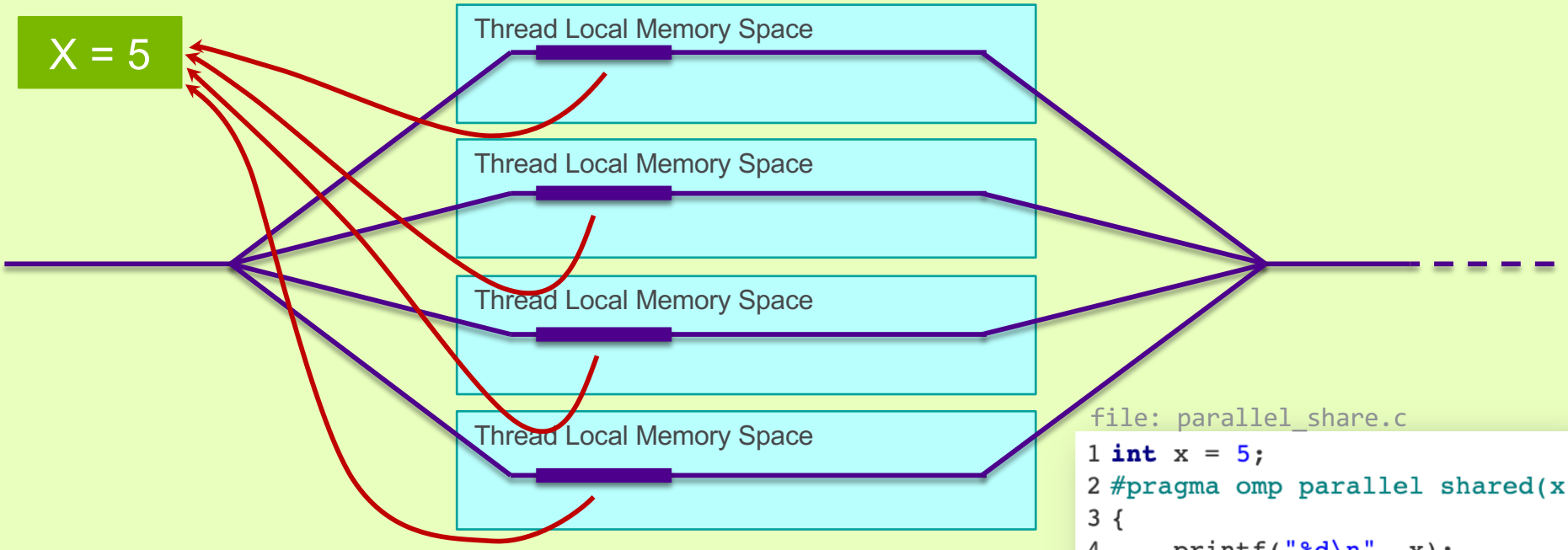
THE OPENMP MEMORY MODEL

Global shared vs thread local memory



OPENMP MEMORY CLAUSES

Shared() clause

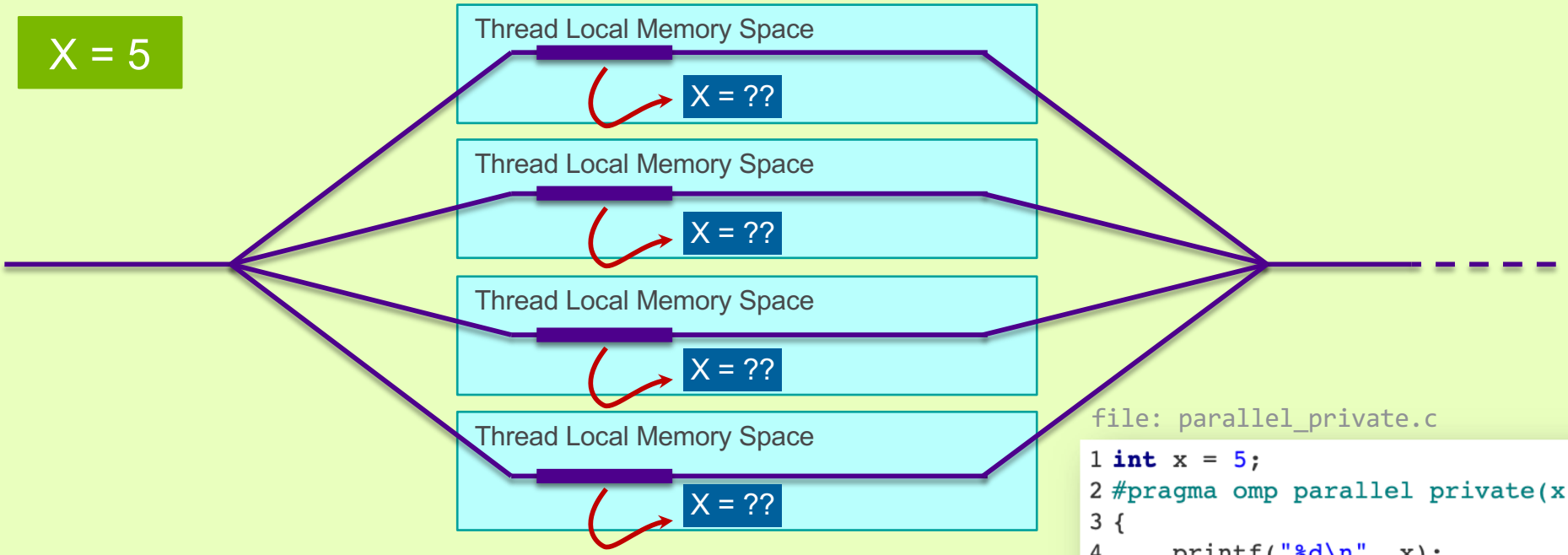


file: parallel_share.c

```
1 int x = 5;
2 #pragma omp parallel shared(x)
3 {
4     printf("%d\n", x);
5 }
```

OPENMP MEMORY CLAUSES

Private() clause

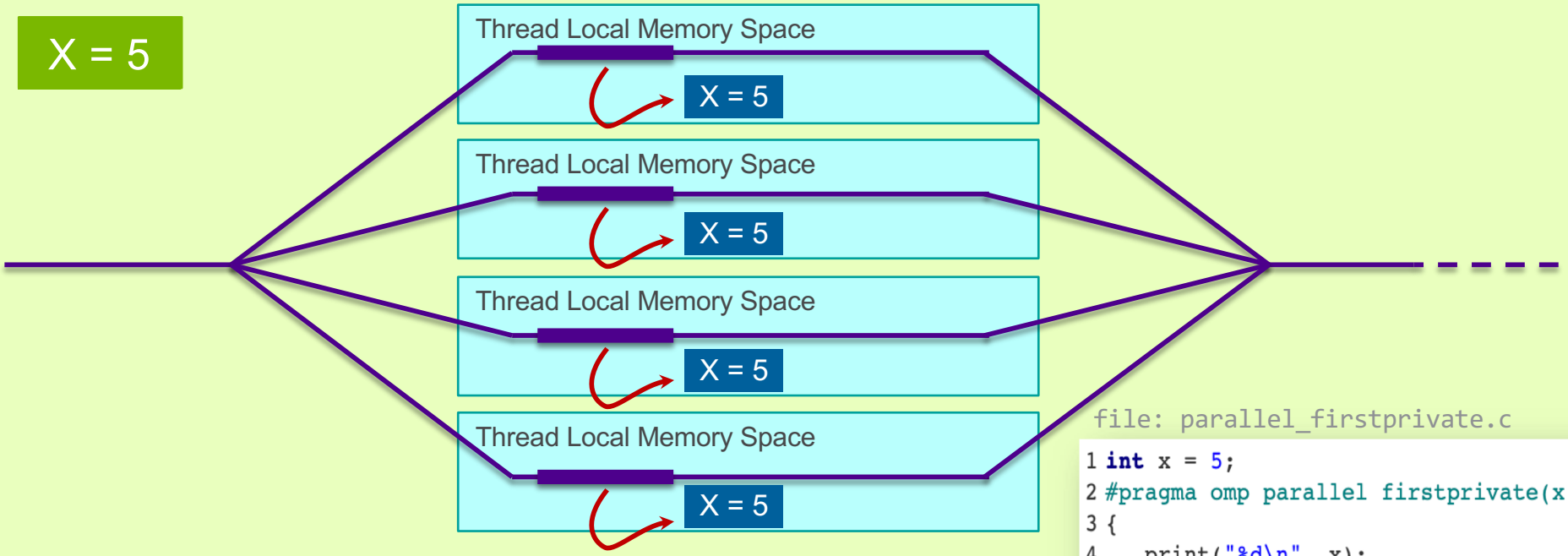


file: parallel_private.c

```
1 int x = 5;
2 #pragma omp parallel private(x)
3 {
4     printf("%d\n", x);
5 }
```

OPENMP MEMORY CLAUSES

firstprivate() clause

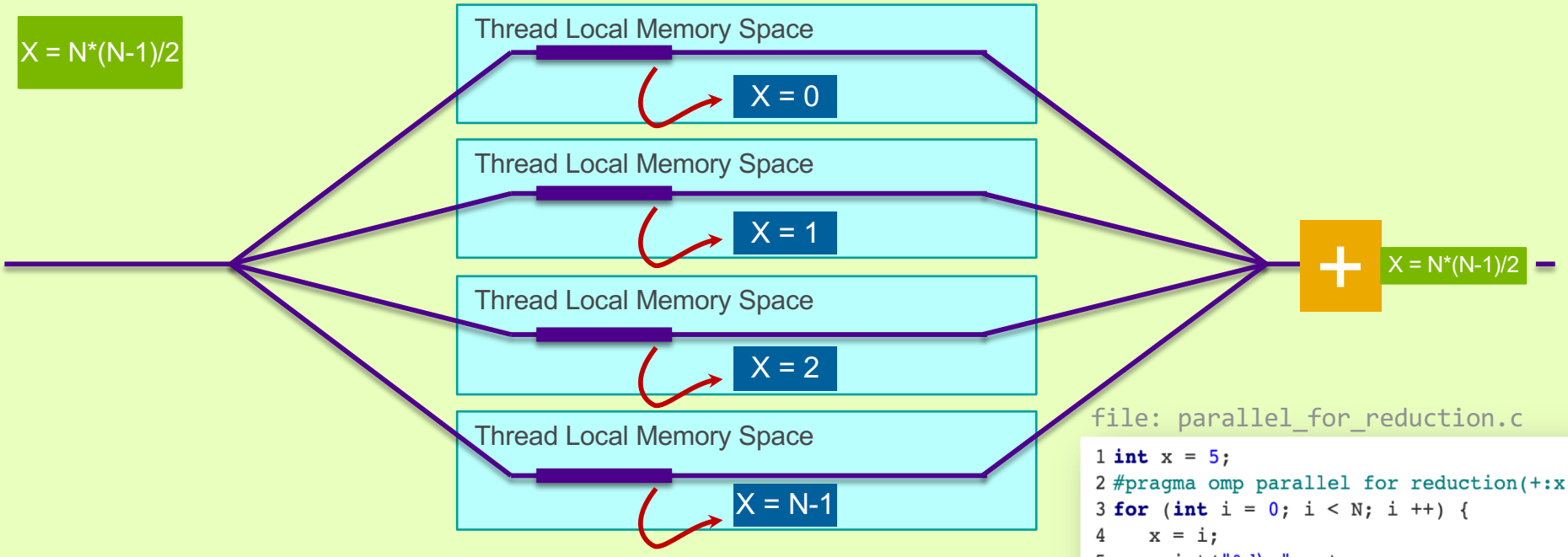


file: parallel_firstprivate.c

```
1 int x = 5;
2 #pragma omp parallel firstprivate(x)
3 {
4    print("%d\n", x);
5 }
```

OPENMP MEMORY CLAUSES

reduction() clause

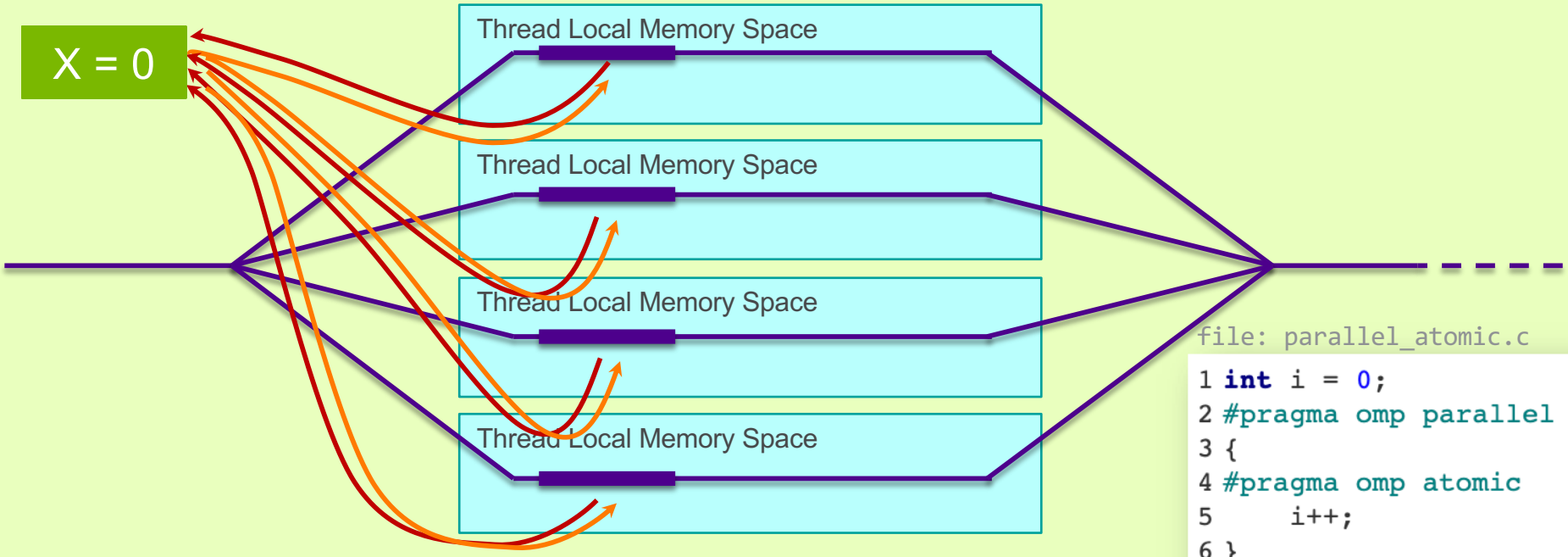


file: parallel_for_reduction.c

```
1 int x = 5;
2 #pragma omp parallel for reduction(+:x)
3 for (int i = 0; i < N; i ++) {
4     x = i;
5     print("%d\n", x);
6 }
```

OPENMP MEMORY CLAUSES

Atomic Directive



file: parallel_atomic.c

```
1 int i = 0;
2 #pragma omp parallel
3 {
4 #pragma omp atomic
5     i++;
6 }
7 printf("i = %d\n", i);
```

TASKING

TASKING IN OPENMP

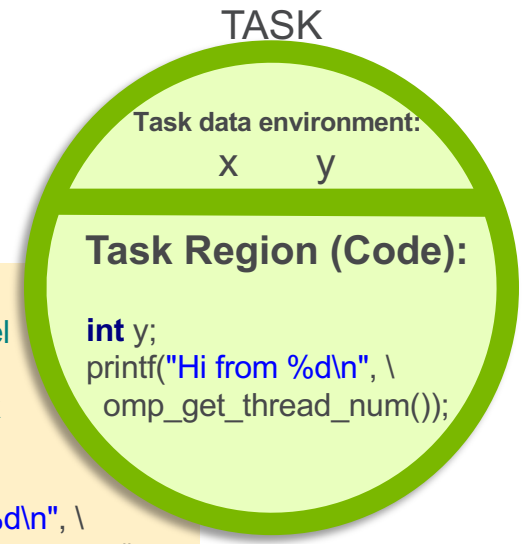
Yet another way of assigning work to threads…

- Before tasking we used worksharing constructs to assign work to threads:
  - For/do loops, sections, single …
- Tasks allow us to create and queue “work” that threads execute
- Additionally it allows controlling dependencies between different work tasks
- We use a **parallel region** to create the threads, and the **tasking constructs** to create work and add it into the work queue

TASKING MODEL

Task definition

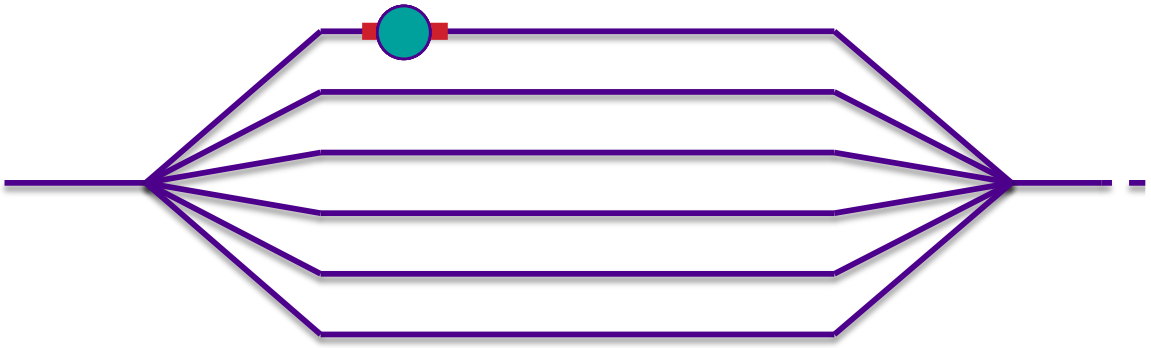
- A task is an instance of executable code and its data environment.
- A task is generated by:
  - `Task`
  - `Taskloop`
  - `Parallel (implicitly)`
  - `Target (implicitly)`
  - `Teams (implicitly)`
- Tasking constructs provide units of work to a thread for execution.

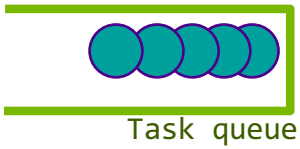



```
int x = 10;
#pragma omp parallel
{
   #pragma omp task
   {
      int y = x;
      printf("Hi from %d\n", \
         omp_get_thread_num();
   }
}
```

TASKING MODEL

Creation of tasks





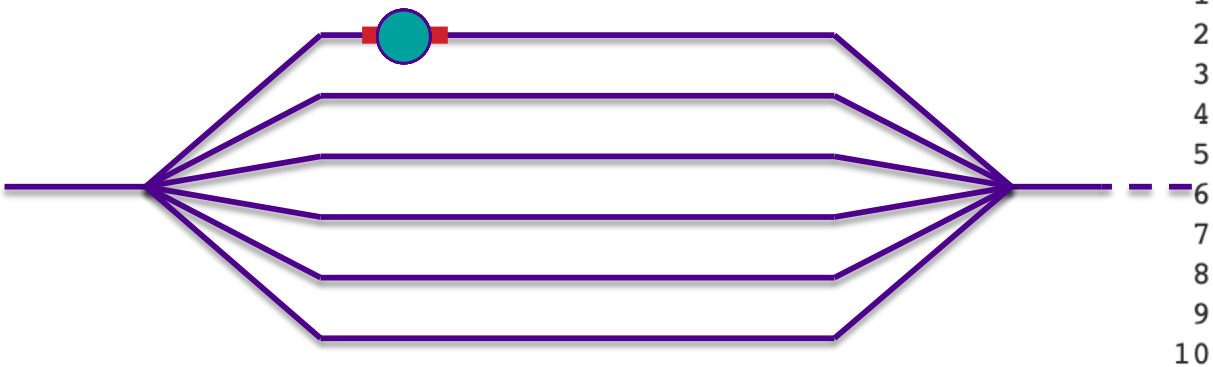
file: tasking.c

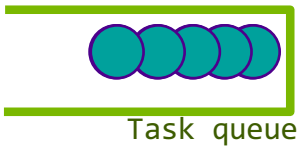
```
1 #pragma omp parallel
2 {
3 #pragma omp master
4   for (int i = 0; i < 5; i++) {
5 #pragma omp task
6     {
7         printf("Hi from %d\n", \
8             omp_get_thread_num();
9     }
10  }
11 }
```

Note: The number of workers is determined by the number of threads in the parallel region

TASKING MODEL

Oversubscription of tasks





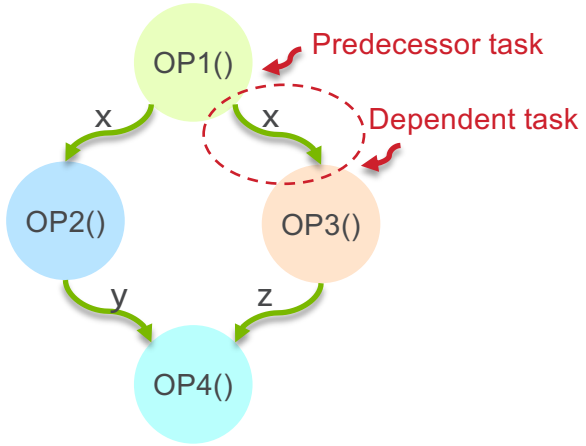

```
1 #pragma omp parallel
2 {
3 #pragma omp master
4   for (int i = 0; i < N; i++) {
5 #pragma omp task
6     {
7       printf("Hi from %d\n", \
8         omp_get_thread_num();
9     }
10  }
11 }
12
```

TASK DEPENDENCIES

Give order to task execution

file: tasking_depend.c

```
1 #pragma omp parallel
2 {
3 #pragma omp single
4   {
5 #pragma omp task depend(out:x)
6     { x = OP1(); }                                   OP1
7 #pragma omp task depend(in:x) depend(out:y)
8     { y = OP2(x); }                                  OP2
9 #pragma omp task depend(in:x) depend(out:z)
10    { z = OP3(x); }                                  OP3
11 #pragma omp task depend(in:z,y)
12    { res = OP4(y,z); }                              OP4
13  }
14 }
```



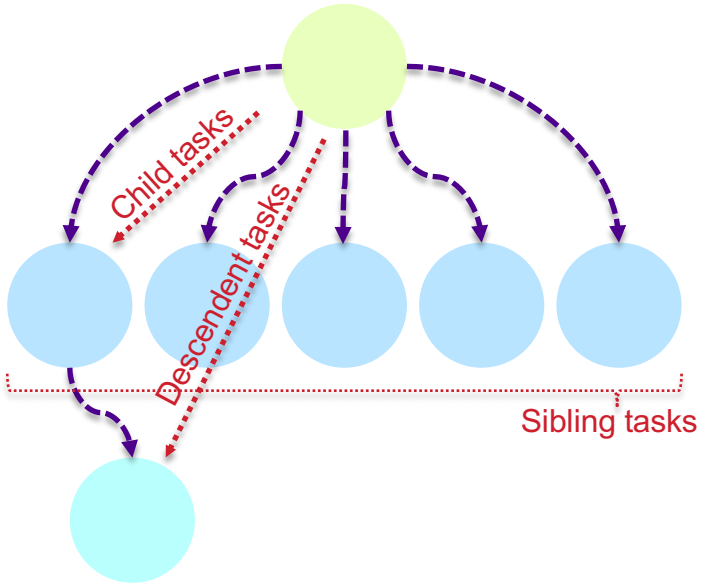
Dependencies guarantees order between tasks if the variable belongs to the same data environment

TASKING MODEL

Terminology

file: tasking_terminology.c

```
1 #pragma omp task                                   T1
2 {
3   for (int i = 0; i < 5; i ++) {
4 #pragma omp task                                   CHILDREN OF T1
5     {
6       printf("I am a child\n")
7       if (i == 0) {
8 #pragma omp task                                   DESCENDENT OF T1
9         { printf("I am a descendent\n"); }
10      }
11    }
12  }
13 }
```

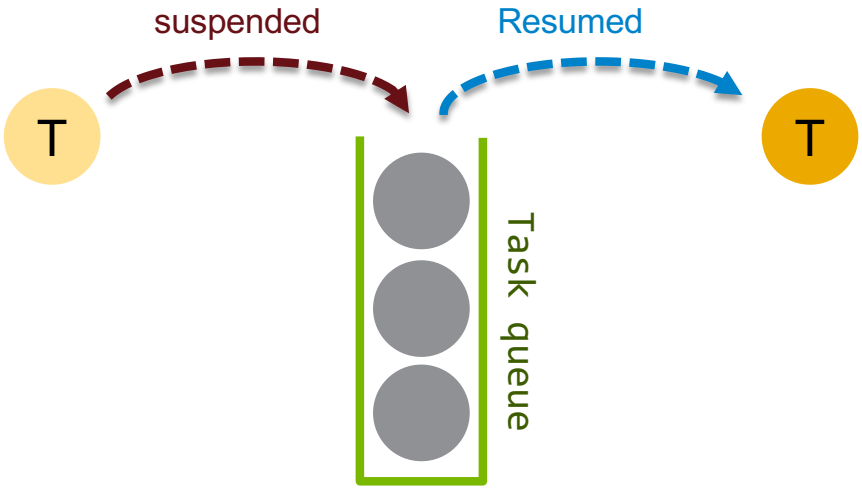


CREATE TASK
*not dependency

TASKING MODEL

Task Scheduling Points

- Task execution can be suspended and resumed later on.
- This can only happen at certain points called **scheduling points.**
  - Some examples:
    - Generation of the task
    - `Taskyield` directive
    - `Taskwait` directive
    - End of `taskgroup` directive
    - Implicit and explicit `barriers`

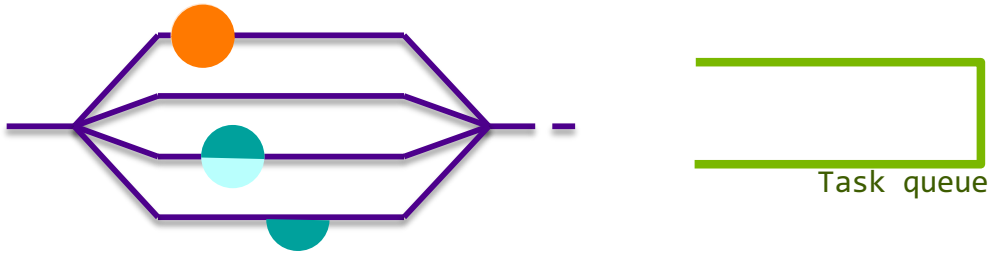


TASKING MODEL

Tied and Untied tasks

file: tasking_untied.c

```
 1 #pragma omp task
 2 {
 3    printf("Hi from %d\n", \
 4      omp_get_thread_num());
 5 #pragma omp taskyeld
 6    printf("Hi from %d\n", \
 7      omp_get_thread_num());
 8 }
 9 #pragma omp task untied
10 {
11    printf("Hi from %d\n", \
12      omp_get_thread_num());
13 #pragma omp taskyeld
14    printf("Hi from %d\n", \
15      omp_get_thread_num());
16 }
```



- **Tied:** Can be resumed only by the same thread that suspended it
- **Untied:** Can be resumed by any thread in the team

TASKING MODEL

Deferred and Undeferred tasks

- Deferring a task means that a task is generated but not executed right away without suspending the execution of the generating (current) task
  - A task is deferred by default
- A non deferred task suspends the execution of the current task until the generated task gets executed

```
1 #pragma omp task
2 {
3   printf("1\n");
4 #pragma omp task
5   {
6     printf("2\n");
7   }
8   printf("3\n");
9 }
```

Deferred

```
1 #pragma omp task
2 {
3   printf("1\n");
4 #pragma omp task if(0)
5   {
6     printf("2\n");
7   }
8   printf("3\n");
9 }
```

Undeferred

file: tasking_undeferred.c

TASKING MODEL

Included, merged, and final tasks

- **Included task:** A task for which execution is sequentially included in the generating task region.
  - **Undeferred** and executed immediately
- **Merged task:** A task for which the **data environment** and **Internal Control Variables** is the same as the generating task
  - Must be **undeferred** and **included**

file: tasking_mergeable.c

- **Final Task:** A task that forces **all of its child tasks** to become **final and included**. Recursively make all descendant tasks included as well
  - It does not merge the tasks, unless allowed by each task (i.e. *mergeable* clause)

TASKING MODEL

Final tasks

```
1 #pragma omp task final(1)
2 {
3     int aNum = 0;
4     for (int i = 0; i < N; i ++) {
5 #pragma omp task private(aNum)
6         { aNum = i; }
7     }
8     printf("aNum = %d\n", aNum);
9 }
```

```
1 #pragma omp task final(1)
2 {
3     int aNum = 0;
4     for (int i = 0; i < N; i ++) {
5 #pragma omp task private(aNum) mergeable
6         { aNum = i; }
7     }
8     printf("aNum = %d\n", aNum);
9 }
```

Equivalent to

Either way

```
1 #pragma omp task final(1)
2 {
3     int aNum = 0;
4     for (int i = 0; i < N; i ++) {
5         int aNum;
6         aNum = i;
7     }
8     printf("aNum = %d\n", aNum);
9 }
```

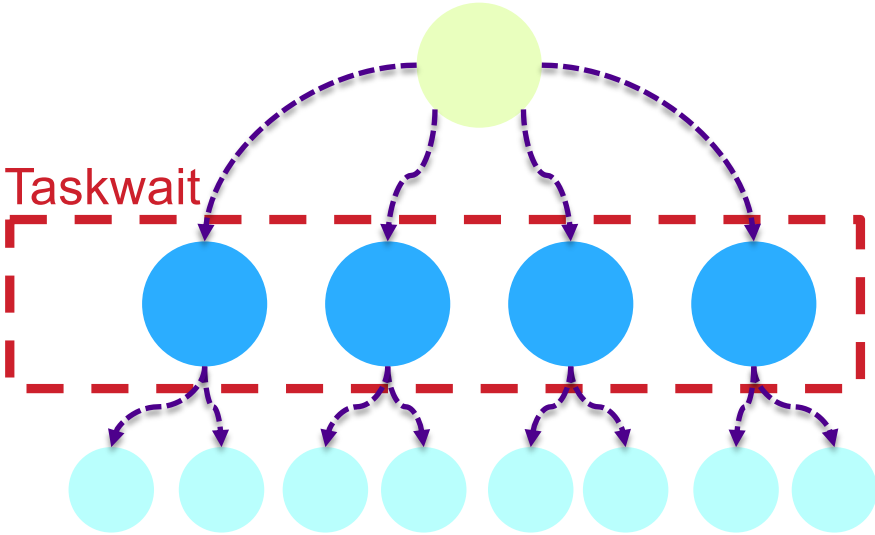
```
1 #pragma omp task final(1)
2 {
3     int aNum = 0;
4     for (int i = 0; i < N; i ++) {
5         aNum = i;
6     }
7     printf("aNum = %d\n", aNum);
8 }
```

TASK SYNCHRONIZATION

Taskwait

file: tasking_taskwait.c

```
1 #pragma omp task
2 {
3   for (int i = 0; i < 4; i ++) {
4 #pragma omp task //Children
5     {
6       printf("Hi there... I'm a child\n");
7       for (int j = 0; j < 2; j++) {
8 #pragma omp task //Descendant
9         {
10          printf("Hi there... I'm a grandchild\n");
11        }
12      }
13    }
14  }
15 #pragma omp taskwait
16  printf("And we're done\n");
17 }
```



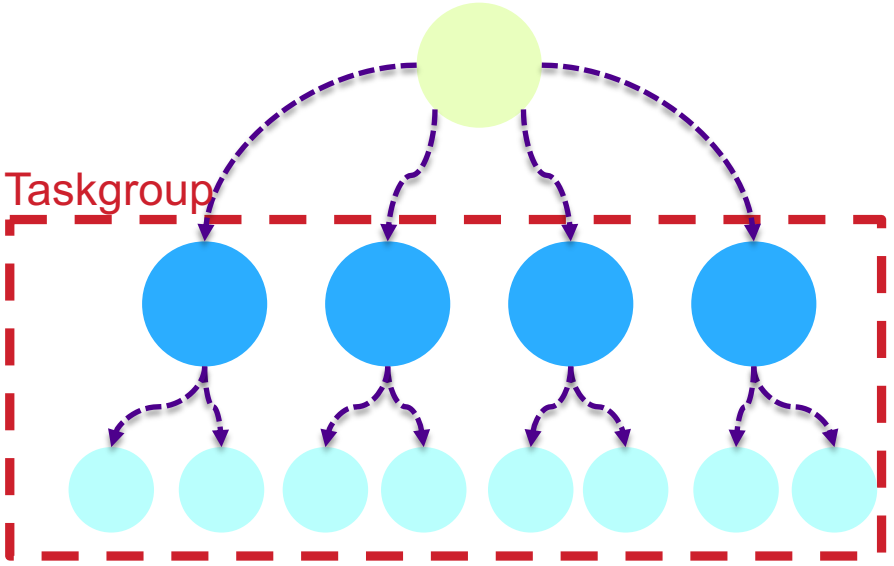
Taskwait yields the current task to wait for the completion of **only the children task**

TASK SYNCHRONIZATION

Taskgroup

file: tasking_taskgroup.c

```
1 #pragma omp task
2 {
3 #pragma omp taskgroup
4   {
5     for (int i = 0; i < 4; i ++) {
6 #pragma omp task //Children
7       {
8         printf("Hi there... I'm a child\n");
9         for (int j = 0; j < 2; j++) {
10 #pragma omp task //Descendant
11          {
12            printf("Hi there... I'm a grandchild\n");
13          }
14        }
15      }
16    }
17  } // Wait for all tasks and descendants
18 }
```



Taskgroup yields the current task to wait for the completion of **all the children task and descendants**

TASK LOOPS

New in OpenMP 4.5

Parallelizing this loop with tasks

```
1 for(int i = 0; i < N; i++) {
2     A[i] = i;
3 }
```

```
1 #pragma omp taskloop grainsize(M)
2 for (int i = 0; i < N; i ++) {
3   A[i] = i;
4 }
```

Taskloops

Allows distributing an iteration loop into multiple tasks

file: tasking_taskloop.c

```
 1 #pragma omp parallel
 2 {
 3 #pragma omp taskgroup
 4   {
 5 #pragma omp single
 6     {
 7       int chunk_size = M, end_chunk;
 8       for (int i = 0; i < N; i += chunk_size) {
 9         end_chunk = (i + chunk_size < N) ? i : N;
10 #pragma omp task firstprivate(i, end_chunk)
11         {
12           for (int j = i; j < end_chunk; j++) {
13             A[j] = j;
14           }
15         }
16       }
17     }
18   }
19 }
```

No taskloops

SUPPORT IN COMPILERS

Implementations are moving fast

Compiler name	OMP Flag	Offloading Flag	Supported Architectures
GCC	-fopenmp	-foffload=<*arch*>=<options>	KNL, NVIDIA, soon-AMD
LLVM	-fopenmp	-fopenmp-target=<*arch*> -Xopenmp-target=<*options*>	NVIDIA
IBM XL	-qsmp=omp	-qoffload	NVIDIA
CRAY CCE	-homp	Not needed	NVIDIA
PGI	-mp		Not supported yet – In progress
Intel	-qopenmp	-qopenmp-offload=<*arch*>	KNL(MIC)
AMD (aomp)	-fopenmp	-fopenmp-target=<*arch*> -Xopenmp-target=<*options*>	NVIDIA, AMD

THE OPENMP SOLLVE TEAM VALIDATION AND VERIFICATION

Help us improve the OpenMP Implementations

Contact information:

Jose Monsalve (josem@udel.edu)
Swaroop Pophale (pophaless@ornl.gov)
Kyle Friedline (utimatu@udel.edu)
Oscar Hernandez (oscar@ornl.gov)
Sunita Chandrasekaran (schandra@udel.edu)

Visit our website
https://crpl.cis.udel.edu/ompvvsollve/

Work supported by the U.S. Department of Energy, Office of Science, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration under contract number DE-AC05-00OR22725.

We also thank all of those who directly or indirectly have help this project.